Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Proyecto de fin de Carrera
Ingeniería en Informática
Curso 2012/2013

Nuevas funcionalidades en la arquitectura Cera-Cranium para el control de un agente inteligente en el entorno del videojuego Unreal Tournament 2004

Carlos Vicente González

Director: Manuel González Bedia
Co-Director: Francisco Serón Arbeloa

Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Junio de 2013

Nuevas funcionalidades en la arquitectura Cera-Cranium para el control de un agente inteligente en el entorno del videojuego Unreal Tournament 2004

RESUMEN

En este proyecto se pretende implementar un agente inteligente (*bot*) para videojuegos de acción en primera persona, que tome decisiones utilizando un controlador basado en una arquitectura cognitiva.

Las arquitecturas cognitivas [1], desde su origen, han buscado modelar el comportamiento y estructura de agentes inteligentes considerados como modelos integrados, es decir, arquitecturas donde se articulen procesos de razonamiento, planificación, aprendizaje, etc. de un modo unificado.

Utilizando como base la arquitectura CERA-CRANIUM[18] se pretende mejorar dicha arquitectura para cumplir la definición de arquitectura cognitiva dada por Vernon, Meta y Sandini [17] y mejorar su valoración en la escala ConScale[10] de evaluación de implementaciones de conciencia artificial.

El entorno de simulación elegido para este proyecto es el videojuego Unreal Tournament 2004 (UT2004). Esta elección vino motivada por la existencia del concurso a nivel mundial "BotPrize", consistente en la implementación de un bot para UT2004 que simule el comportamiento humano. Las herramientas utilizadas son el videojuego UT2004 junto con la ampliación Gamebots 2004 (que permite ejecutar bots en el videojuego) y la plataforma Pogamut 3, que permite programar al agente virtual en el lenguaje de programación Java, y conectarlo y recibir información del videojuego.

De entre las nuevas implementaciones realizadas podríamos destacar la integración de un sistema de toma de decisiones para controlar el funcionamiento del bot basado en Soar[2] que reproduce mecanismos de cambio de estrategia ante el reconocimiento de modificaciones en el entorno, aprendiendo de su propia experiencia mediante "aprendizaje por refuerzo" así como una mejor simulación del comportamiento humano gracias a la implementación de algunos comportamientos de inspiración emocional.

Para comprobar la efectividad de estas mejoras, se ha preparado un experimento siguiendo los estándares del concurso "BotPrize".

Para Laura,

aunque no entienda absolutamente nada.

Agradecimientos

A mi familia y amigos, por su apoyo y comprensión durante todos estos años, especialmente a Mateo, Alfonso y Mario por continuar a mi lado cuando esta aventura acabe.

A Manuel y Paco por darme la oportunidad de trabajar en este proyecto y a David, Dani y Carlos por haberme ayudado a sobrellevar el proceso y haber colaborado como conejillos de indias.

Y por último, aunque no por ello menos importante, a Alan Turing. Sin él nada de esto sería posible.

Muchas gracias a todos.

Índice general

Índice de figuras

Índice de tablas

Parte I

Memoria

Capítulo 1

Introducción

El objetivo de este proyecto es el desarrollo de un agente autónomo virtual (*bot*, ver Glosario) para videojuegos de acción en primera persona (FPS) y más en concreto para el entorno de simulación Unreal Tournament 2004. Dicho comportamiento "estará controlado" por una versión mejorada de la arquitectura CERA-CRANIUM[18], basada en la Teoría del Espacio de Trabajo Global[19], tratando de alcanzar un nivel superior en la escala de medición de la conciencia artificial ConScale[10] y específicamente, aumentar de un nivel de conciencia "reactivo" (nivel 2) a uno "adaptativo" (nivel 3) así como dotar al sistema de algunas de las características de los niveles posteriores "atencional", "ejecutivo" y "emocional" (4, 5 y 6).

Entre las nuevas implementaciones se encuentran:

1. mecanismos de evaluación de propio rendimiento en la consecución de múltiples objetivos (memoria medio-largo plazo) y del estado global propio
2. representación de los efectos de este estado en el comportamiento (sentimientos)
3. cambio de contexto entre los objetivos perseguidos

Además de la evaluación teórica realizada en base a ConScale, se realizará una evaluación práctica mediante un experimento controlado en el que se analizará la "humanidad" (en términos de los criterios del concurso BotPrize) que otorgan jugadores reales al comportamiento de este nuevo *bot* en comparación con su versión previa.

El videojuego Unreal Tournament 2004, es un juego de código abierto que dispone de un conjunto de librerías que permiten la creación de bots de forma independiente al entorno gráfico, dicho conjunto de librerías se llaman Pogamut[3](versión 3.4) y están escritas en Java.

1.1. Objetivos

El proyecto fin de carrera que se describe en este documento tiene los siguientes objetivos:

- Evaluación de la arquitectura CERA-CRANIUM según la definición de Vernon, Meta y Sandini[17] sobre una arquitectura cognitiva.
- Evaluación del bot CCBot2 en la escala ConScale de habilidades cognitivas.
- Implementación de nuevas funcionalidades utilizando como base las dos evaluaciones anteriores, las posibles carencias y las potenciales mejoras de CERA-CRANIUM y el CCBot2.

- Analizar experimentalmente el comportamiento del nuevo bot.
- Obtener conocimientos y experiencia en el uso de tecnologías y herramientas para el desarrollo software en la ingeniería, como pueden ser Java[1], SVN[2], así como aprender a trabajar con herramientas desconocidas para mí como son Pogamut, jSoar o los lenguajes que utilizan las diferentes arquitecturas cognitivas; y otras que conozco pero no utilizo habitualmente como los sistemas basados en reglas.
- Introducirse en el ámbito de la investigación de la inteligencia artificial utilizando datos de otros ámbitos relacionados,como la psicología.

1.2. Motivación

Han sido muchos los motivos que me han llevado a la elección del desarrollo de un sistema de toma de decisiones que controle el comportamiento de un agente autónomo virtual (bot) del videojuego Unreal Tournament 2004 como mi Proyecto Fin de Carrera. De entre todos ellos los principales han sido:

- La existencia de un concurso a nivel mundial "BotPrize"[3](Anexo B) que consiste en la implementación de un bot para el Unreal Tournament 2004 que simule el comportamiento humano.
- La posibilidad de profundizar en una arquitectura ganadora del "BotPrize" como es CERACRANIUM, su paradigma cognitivo y su funcionamiento.
- Me brindaba la posibilidad de adentrarme en el mundo de la inteligencia artificial, los videojuegos y la psicología, ya que tenía interés en temas de emociones, aprendizaje, motivación o las arquitecturas cognitivas, y son temas que no se tratan en profundidad en la carrera.
- Me ofrecía una forma interesante de adquirir conocimientos y experiencia en herramientas y tecnologías como Java y también en herramientas completamente desconocidas como Pogamut y jSoar. Este ha sido un proyecto donde he podido poner en práctica muchos conocimientos adquiridos durante la carrera, así como abordar el uso de nuevas herramientas y tecnologías desconocidas.
- Antes de definir mi futuro laboral quería tener una experiencia más ligada al mundo de la investigación que al mundo empresarial.
- Por último, existe la posibilidad de que mi trabajo ayude y sea utilizado por otras personas: Unido al desarrollo de este proyecto se ha propuesto la creación de un concurso de características similares a BotPrize a nivel universitario que se desarrollará durante el curso 2013/2014. La nueva versión del bot participa además en un estudio comparativo realizado por el profesor Juan Peralta de la Universidad Autónoma de Barcelona en una versión preliminar de lo que será este "BotPrize universitario".

1.3. Tecnologías utilizadas

Para la implementación de los bots se ha utilizado el videojuego Unreal Tournament 2004 junto con la ampliacion GameBots2004 (que permite ejecutar bots en el videojuego) y la plataforma

[1] http://www.java.com
[2] http://es.wikipedia.org/wiki/Subversion
[3] http://botprize.org/

Pogamut 3 que permite programar al agente virtual en el lenguaje de programacion Java, conectarlo y recibir informacion del videojuego mediante un plugin para Netbeans (ver Anexo A).

El programa UnrealED, incluido en la instalacion del videojuego, ha sido utilizado para el diseño de un mapa expecial para el entrenamiento del bot.

Por último, se ha utilizado JSoar para desarrollar el sistema de toma de decisiones. JSoar es una implementación en Java de la arquitectura cognitiva Soar en forma de biblioteca para la construcción de sistemas basados en agentes.

Para más información sobre las tecnologías utilizas en este proyecto y las ventajas e inconvenientes encontradas durante el desarrollo del proyecto derivadas de dichas tecnologías consultar el Anexo F.

1.4. Contenido de la memoria

A continuación se presenta un resumen de los contenidos de esta memoria:

- El capítulo 1 es en el que nos encontramos y sirve de introducción al resto de la documentación.
- El capítulo 2 trata sobre el estado del arte del Proyecto y en él se analiza la definición de arquitectura cognitiva de Vernon, Meta y Sandini [17] así como una clasificación de algunas de las arquitecturas cognitivas más importantes en la actualidad.
- En el capítulo 3 se encuentra la descripción de la estructura y funcionamiento de la arquitectura cognitiva CERA-CRANIUM y de su implementación CCBot2 así como la evaluación de ambas en función de la definición dada en el capítulo 2 y de la escala ConScale de habilidades cognitivas.
- En el capítulo 4 se explican las nuevas implementaciones desarrolladas en el bot CCBotSoar (Memoria de Objetos, Sistema de Toma de Decisiones, Mapa de Peligro y Modelo de Riesgo) y su integración en el sistema así como una nueva análisis del bot en la escala ConScale para comprobar el avance producido gracias a estas nuevas implementaciones.
- El capítulo 5 contiene un breve resumen del entrenamiento recibido por CCBotSoar, asi como la definición y resultados de los experimentos realizados.
- En el capítulo 6 se muestra una conclusión general sobre el desarrollo del proyecto, posibles líneas futuras y valoración personal del proyecto.
- Por último, encontraremos las siguientes secciones:
  - Glosario: Contiene la definición de algunos términos mencionadas en la memoria.
  - Bibliografía: Fuentes de información consultadas para realizar el proyecto.
  - Anexos: Información adicional del proyecto referenciada desde apartados anteriores. Incluye el manual de Pogamut, información sobre el Bot Prize, ConsScale, un resumen de las clases y métodos de la implementación CCBotSoar y las tecnologías utilizadas en el desarrollo de este proyecto.

1.5. Planificación

Durante los meses de duración del proyecto, se han realizado labores de documentación y formación en las tecnologías y plataformas usadas, prestando especial atención a Soar, CERA-CRANIUM, ConScale y las distintas definiciones aceptadas de una arquitectura cognitiva así como la implementación CCBot2. Además, se han investigado diferentes tipos de recursos (redes neuronales, sistemas de razonamiento basado en casos, modelos de infotaxis, etc) que, aunque finalmente no se utilizaron, fue necesario conocer no sólo para barajar varias opciones sino también para enriquecer el planteamiento de las nuevas implementaciones. El diagrama de Gantt correspondiente a la realización del proyecto se muestra en la figura 1.1.

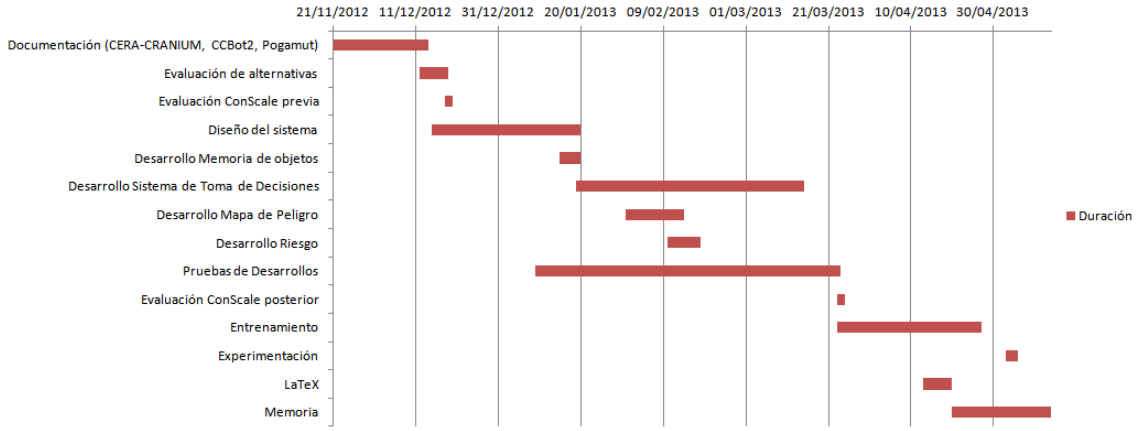

Figura 1.1: Diagrama Gantt del proyecto

Capítulo 2

Estado del arte: Definición y análisis de arquitecturas cognitivas

En este capítulo se presenta una definición de arquitectura cognitiva basada en la comparación de dos paradigmas (emergente y cognitivista) y se analizan algunas de las arquitecturas cognitivas actuales más importantes con el fin de enriquecer dicha definición.

2.1. Definición de arquitectura cognitiva

Se puede definir una arquitectura cognitiva como el esquema o patrón para estructurar los elementos funcionales que configuran a un agente inteligente. Las arquitecturas cognitivas proponen procesos computacionales que actúan como ciertos sistemas cognitivos (como podrían ser un "ser humano") o, más a menudo, actos inteligentes bajo determinada definición. El término arquitectura implica un enfoque que intenta modelar no solo el comportamiento, sino también las propiedades estructurales del sistema modelado.

Las arquitecturas cognitivas existentes pueden clasificarse en dos grupos principales:

- Por un lado, la aproximación **cognitivista**, basada en información simbólica procesada por sistemas representacionales[14]. Las aproximaciones cognitivistas se corresponden con la clásica y aún común visión de que "la cognición es un tipo de computación simbólica, racional, encapsulada, estructurada y algorítmica" y de que un sistema cognitivo "instancia estas representaciones físicas como códigos cognitivos y sus comportamientos son una consecuencia causal de las operaciones que acarrean estos códigos" [21].
- Por otro los sistemas conexionistas, dinámicos y enactivos (todos ellos basados en una mayor o menor extensión de los principios de auto-organización [20]) agrupados bajo el nombre general de sistemas **emergentes**. Estos sistemas discuten contra la visión del procesado simbólico y se posicionan a favor de tratar la cognición como emergente, auto-organizada y dinámica [22, 23].
- Existe además un tercer grupo denominado de arquitecturas denominadas híbridas que combinan las características de ambos paradigmas.

En el Cuadro 2.1 se desarrolla la comparación entre los paradigmas cognitivista y emergente en función de doce características distintas: **Operatividad computacional, Infraestructura de la**

representación, Grounding semántico, restricciones temporales, Epistemología global, Corporeidad, Percepción, Acciones, Anticipación, Adaptación, Motivación y Relevancia de la autonomía.

En el paradigma cognitivo, el foco de atención en una arquitectura cognitiva se centra en los aspectos en los que la cognición es constante a través del tiempo y que resulta relativamente independiente de la tarea [34, 31, 32]. Ya que las arquitecturas cognitivas representan la parte fija de la cognición, no pueden conseguir nada de su funcionamiento interno y necesitan adquirir o ser provistas de conocimiento para realizar cualquier tarea. Esta combinación de una **arquitectura cognitiva** dada y una serie de **conocimientos particulares** se conoce habitualmente como **modelo cognitivo**. En la mayoría de los sistemas cognitivistas el conocimiento incorporado al modelo se determina por el diseñador humano, aunque existe un incremento en el uso del machine learning para aumentar y adaptar este conocimiento.
La especificación de una arquitectura cognitiva consiste en sus conjeturas representacionales, las características de sus recuerdos, y los procesos que operan con esos recuerdos. La arquitectura cognitiva define la manera en la que un agente cognitivo gestiona la eliminación de sus fuentes [33]:

- Para las aproximaciones cognitivistas, estas fuentes son el sistema computacional en el que se realiza el procesado de símbolos físicos. La arquitectura especifica los formalismos para la representación del conocimiento y la memoria utilizada para almacenarlos, los procesos que actúan sobre ese conocimiento y los mecanismos de aprendizaje que se adquieren. Habitualmente, también tiene la posibilidad de programar el sistema para que los sistemas inteligentes puedan ser instanciados en una especie de dominio de aplicación [34].

- Para las aproximaciones emergentes, la necesidad de identificar una arquitectura surge de la complejidad intrínseca que caracteriza a un sistema cognitivo, y la necesidad de dotar de algún tipo de estructura en la que enmarcar los mecanismos de percepción, acción, adaptación, anticipación, y motivación que permiten que el desarrollo ontogenético durante el ciclo de vida del sistema. Es esta complejidad la que distingue una arquitectura cognitiva emergente de un simple control robótico conexionista que habitualmente aprenden asociaciones mediante tareas específicas, por ejemplo, la red auto-organizativa de Konohen citada en [35]. En cierto modo, la arquitectura cognitiva de un sistema emergente se corresponde con las capacidades innatas de las que está dotada por la filogénesis del sistema, las cuáles no debe aprender pero que, por supuesto, puede evolucionar. Estas capacidades básicas facilitan la ontogénesis del sistema, representan el punto de partida inicial para el sistema cognitivo y aseguran la base y los mecanismos para su consiguiente desarrollo autónomo, un desarrollo que influirá directamente en la propia arquitectura.

2.2. Arquitecturas actuales

La Figura 2.1 muestra un resumen de todas las arquitecturas en las que se han analizado una a una las doce características de los sistemas cognitivos que se describieron anteriormente. Se han omitido las cinco primeras características – Operatividad computacional, Infraestructura de representación, Nociones semánticas, Restricciones temporales y Epistemología global – porque se pueden inferir directamente desde el propio paradigma en el que el que se basa el sistema: cognitivista (C), emergente (E) o híbrido (H).

Una "$\times$" indica que esta característica es clave en la arquitectura, "+" indica que existe esta característica y un espacio en blanco indica que esta característica no está presente en la arquitectura. Se asigna una "$\times$" en la columna de Adaptación si y solo si el sistema es capaz de desarrollar (en el sentido de crear nuevos marcos o modelos de representación) y no simplemente aprender (en el sentido de estimar parámetros de un modelo) [36].

Architecture	Paradigm	Embodiment	Perception	Action	Anticipation	Adaptation	Motivation	Autonomy
Soar	C				+	+		
Epic	C		+	+	+			
ACT-R	C		+	+	+	+		
ICARUS	C		+	+	+	+		
ADAPT	C	×	×	×	+	+		
AAR	E	×	×	×			+	×
Global Workspace	E	+	+	+	×		×	×
I-C SDAL	E	+	+	+	+	+	×	×
SASE	E	×	×	×	+	×	×	×
Darwin	E	×	×	+		×	×	×
HUMANOID	H	×	×	×	×	+	+	
Cerebus	H	×	×	×	+	+		
Cog: Theory of Mind	H	×	×	×	+			
Kismet	H	×	×	×			×	

Figura 2.1: Comparativa de arquitecturas (extraído de [17])

2.3. Conclusiones

- Un sistema cognitivo de desarrollo estará constituido por una red de subsistemas multifuncionales distribuidos que cooperan y compiten, cada uno de los cuales posee su propia infraestructura de codificación o representación, que alcanzan juntos un objetivo común de comportamiento efectivo controlados por algún tipo de mecanismo de auto-sincronización o circuito de modulación. Esta red forma la configuración filogenética y sus habilidades innatas.
- Una arquitectura cognitiva de desarrollo debe ser capaz de adaptarse mediante el aprendizaje y, aún más importante, a través de la modificación de gran parte de la estructura y la organización del propio sistema de manera que sea capaz de alterar sus situaciones dinámicas basándose en la experiencia, para expandir su repertorio de acciones y adaptarse a nuevas circunstancias. Este desarrollo debe estar controlado por factores tanto explorativos como sociales, el primero referido al descubrimiento de novedades en el mundo exterior y el potencial de las propias acciones del sistema, y el segundo con la interacción entre agentes, las actividades compartidas, y patrones mutuamente construidos mediante comportamiento compartido. Existe una gran variedad de paradigmas de aprendizaje que se pueden utilizar para un desarrollo efectivo, incluyendo, pero no necesariamente limitado a, **aprendizaje sin supervisión**, **aprendizaje por refuerzo** y **aprendizaje surpevisado**.
- Puesto que los sistemas cognitivos no son sólo adaptativos sino además anticipativos, es crucial que tenga (filogenia) o desarrolle (ontogenia) algún mecanismo para ensayar escenarios hipotéticos y un mecanismo que module el comportamiento actual del sistema.

Característica	Cognitivista	Emergente
Operatividad computacional	Manipulación basada en reglas (por ejemplo, procesamiento sintáctico) de tokens simbólicos, habitualmente de manera secuencial.	Explotan procesos de auto organización, producción, mantenimiento y desarrollo, a través de la interacción concurrente de una red de componentes distribuidos.
Infraestructura de representación	Patrones de tokens simbólicos para referirse a eventos del mundo exterior. Producto descriptivo de un diseñador humano, normalmente puntual y local.	Sistema global de estados codificados .
Grounding semántico	Identificación simbólica de perceptos ya sean descritos por el diseñador o aprendidos por asociación.	Construcción adaptativa de comportamientos.
Restriccciones temporales	No existe.	Sincronizadamente y en tiempo real con los eventos del entorno.
Epistemología global	Cada agente está enmarcado en un entorno cuya estructura semántica es independiente de la cognición del sistema.	Dependiente del historial de experiencias del agente.
Corporeidad	Gracias a sus raíces funcionalistas (la cognición es independiente de la plataforma física en la que está implementada[27]), no implica corporeidad.	Intrínsecamente unidos a su representación física [26, 24, 25].
Percepción	Interfaz. Representaciones simbólicas abstractas del mundo externo desde los datos sensoriales.	Cambio en el estado del sistema en respuesta a perturbaciones en el entorno para mantener la estabilidad del propio sistema.
Acciones	Consecuencias causales del procesamiento simbólico de las representaciones internas.	Perturbaciones realizadas en el entorno por el propio sistema.
Anticipación	Planificación utilizando algún tipo de razonamiento procedural o probabilístico con un modelo *a priori*.	Requiere que el sistema visite un determinado números de estados en su espacio auto-construido de estados percepción-acción sin comprometer las acciones asociadas.
Adaptación	Adquisición de nuevo conocimiento	Alteración estructural o reorganización para que se hagan efectivas nuevas situaciones dinámicas. [30].
Motivación	Resuelven los puntos muertos (*impasses*) [28].	Agrandan el espacio de interacción [29].
Relevancia de la autonomía	No necesariamente implementada.	Crucial. La cognición implica autonomía.

Tabla 2.1: Comparativa entre paradigmas (extraído de [17])

Capítulo 3

CERA-CRANIUM y CC-Bot2

El presente capítulo muestra la estructura y funcionamiento de la arquitectura CERA-CRANIUM y del CC-Bot2, implementación basada en dicha arquitectura, sendos puntos de partida de la investigación de este proyecto.

3.1. CERA-CRANIUM

CERA-CRANIUM [6] es una arquitectura cognitiva, diseñada para controlar augentes autónomos (tanto robots físicos como bots para videojuegos) basada en un modelo computacional de la consciencia. La principal inspiración de CERA-CRANIUM es la Teoría del Espacio de Trabajo Global (Global Workspace) [19], un tipo de espacio de memoria compartida donde diferente agentes – en este caso representados por procesadores especializados – pueden colaborar y competir con otros dinámicamente.

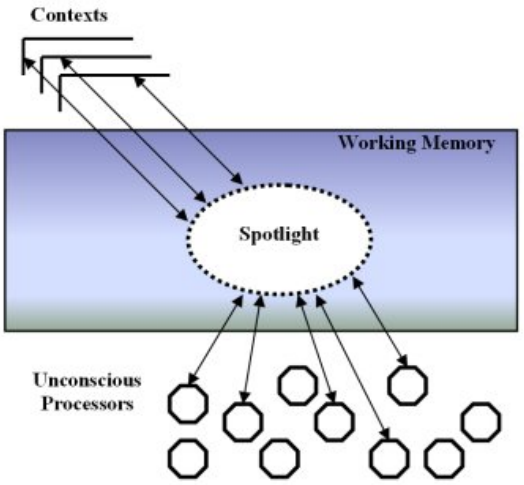

Figura 3.1: Modelo del Espacio de Trabajo Global (extraído de [18])

CERA-CRANIUM está compuesto por dos componentes principales (ver Figura 3.1):

• **CERA**: arquitectura de control estructurada en capas.

• **CRANIUM**: herramienta para la creación y gestión de grandes cantidades de procesos en espacios de trabajo compartidos.

Como se explica más abajo, CERA utiliza los servicios de CRANIUM con el objetivo de generar procesos de percepción dinámicos y adaptables orquestados por un modelo computacional de la consciencia. En términos de control del bot, CRANIUM proporciona un mecanismo para sincronizar un número de procesadores especializados diferentes que funcionan concurrentemente. Estos procesadores pueden ser de muchos tipos, aunque normalmente son detectores de información sensorial o generadores de comportamiento.

Siguiendo la arquitectura de orientación a servicios de MRDS (Microsoft Robotics Developer Studio), los componentes de CERA y CRANIUM se integran como servicios ligeros distribuidos. MRDS incorpora un sistema de ejecución de tiempo real llamado CCR (Concurrency and Coordination Runtime) que está basado en .NET Framework. Partes de CERA y de CRANIUM se basan en el CCR para soportar patrones de coordinación complejos y asíncronos. También se usan las colas de hilos de alto rendimiento para ejecutar un gran número de procesadores inconscientes y maximizar la concurrencia. El protocolo DSSP (Decentralized Software Services Protocol) hace posible la composición de estos servicios. El modelo de servicios DSS permite la interacción de los servicios en un entorno descentralizado, por lo tanto el problema de la capacidad limitada de CPU en el robot desaparace.

CERA

CERA (Conscious and Emotional Reasoning Architecture) es una arquitectura cognitiva distribuida en capas diseñada para implementar un sistema de control para agents autónomos. La actual definición de CERA está estructurada en cuatro capas (Figura 3.2): la **capa de servicios senso-motores**, **capa física**, **capa específica de misión** y **capa núcleo**. Aunque las capas superiores tienen asignados significados más abstractos, la definición de las capas en CERA no está directamente asociada a comportamientos específicos. En su lugar, gestiona cualquier procesador especializado que opere en el tipo de representación utilizado en un nivel particular, como por ejemplo la capa física con representaciones cercanas a los datos sensoriales o la capa misión con representaciones de más alto nivel y orientadas a tareas concretas.

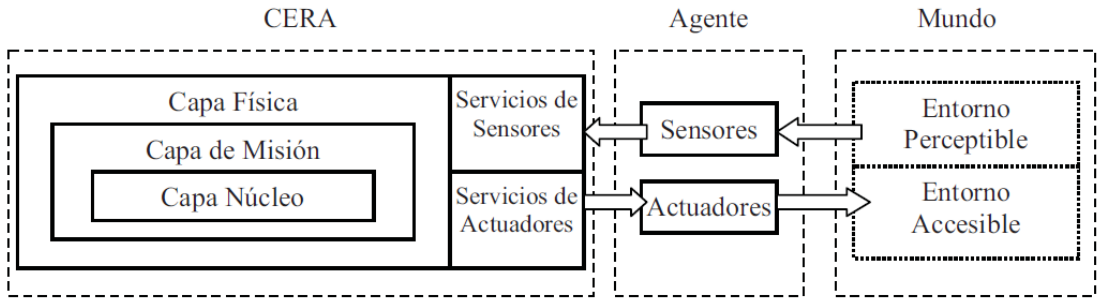

Figura 3.2: Diseño estructurado en capas de la arquitectura cognitiva CERA (extraído de [18])

- La capa CERA de **servicios sensorio-motores** consta de una serie de servicios de interacción y comunicación que implementan el acceso requerido tanto a los sensores de lectura como a los comandos de actuación. Estos servicios proveen a la capa física con una interfaz de acceso uniforme a la maquinaria física (tanto virtual como simulada) del agente.

- La capa **física** de CERA engloba los sensores y actuadores de representación a bajo nivel del agente. Los mecanismos análogos están implementados a este nivel con comandos de actuación, asegurando que los parámetros de los comandos se encuentran entre unos límites seguros.

- La capa **específica de misión** produce y gestiona contenido sensorio-motor elaborado relacionadocon los comportamientos vitales del agente y sus misiones particulares. En esta estapa los contenidos simples que la capa física on combinados con contenidos más complejos y de significado específico, relacionados con los objetivos del agente. La capa específica de misión puede ser modificada independientemente del resto de capas CERA de acuerdo con las tareas asignadas y las necesidades del agente.

- La capa **núcleo** de CERA es el control a más alto nivel y se compone de una serie de módulos que simulan funciones cognitivas de alto nivel. La definición e interacción entre estos módulos puede ser ajustada para implementar un modelo cognitivo concreto, regulando la manera en la que trabajan las capas más bajas de CERA.

La **capa física** y la **capa específica de misión** están inspiradas en teorías cognitivas de la consciencia, donde grandes conjuntos de procesos paralelos compiten y colaboran en un espacio compartido de trabajo buscando una solución global. De hecho, un agente controlado por CERA está dotado con dos espacios de trabajo jerarquizados que funcionan coordinadamente con el objetivo de encontrar dos soluciones globales e interconectadas: una está relacionada con la percepción y la otra con la acción.

CRANIUM

CRANIUM proporciona un subsistema en el cual CERA puede ejecutar varios procesos concurrentes coordinados pero asíncronos que compiten entre ellos. Cada uno de estos **procesadores especializados** están diseñados para desarrollar una función específica con tipos de información concretos. En cualquier momento, el **nivel de activación** de un procesador concreto se calcula basado en una estimación heurística de cuánto puede contribuir a la solución global que actualmente se encuentra en el ETC. Los parámetros concretos usados para esta estimación se establecen en la capa de núcleo CERA. Por norma general, los ETC de CRANIUM operan constantemente ajustados por comandos enviados desde la capa núcleo de CERA. Este mecanismo cierra los bucles de realimentación entre la capa núcleo y el resto de la arquitectura: la entrada de la capa núcleo (percepción) se regula en función de su propia salida (modulación del ETC), que al volver determina qué se percibe.

3.2. CC-Bot2

CC-Bot2 es una implementación Java de la arquitectura CERA-CRANIUM especialmente desarrollada para la competición 2K BotPrize. Esta implementación tiene las siguuientes características:

- La capa CERA de servicios sensorio-motores es básicamente una capa de **adaptación a Pogamut 3**.
- La representación usada para los datos sensoriales y los comandos en la capa física, como podrían ser “aparece un jugador en mi campo de visión” o “estoy siendo atacado”, es **Pogamut 3**

- Puesto que la implementación está hecha para participar únicamente en partidas Death-Match, la **misión es relativamente simple**: matar más y morir menos. Los perceptos que maneja la capa de misión son del tipo "este jugador es mi enemigo" o "el enemigo X me está atacando").

- La capa núcleo contiene el **código para el mecanismo de atención** (otros módulos pueden ser añadidos en un futuro).

- CRANIUM está basado en un gestor de tareas que crea dinámicamente **nuevos hilos de ejecución** para cada procesador activo.

- En el CC-Bot2 usamos **dos ETC** de CRANIUM separados pero conectados e integrados con la arquitectura CERA:

  1. El ETC de nivel más bajo está localizado en la capa física de CERA, donde los procesadores especializados son alimentados con datos provenientes de los servicios sensores de CERA (Pogamut).
  2. El segundo ETC, localizado en la capa de misión específica de CERA, contiene procesadores especializados de más alto nivel que tienen como entrada la información que proviene de la capa física o información generada en el propio ETC 3.3.

  La información perceptual que fluye está organizada en paquetes llamados **perceptos simples**, **perceptos complejos** y **perceptos de misión**.

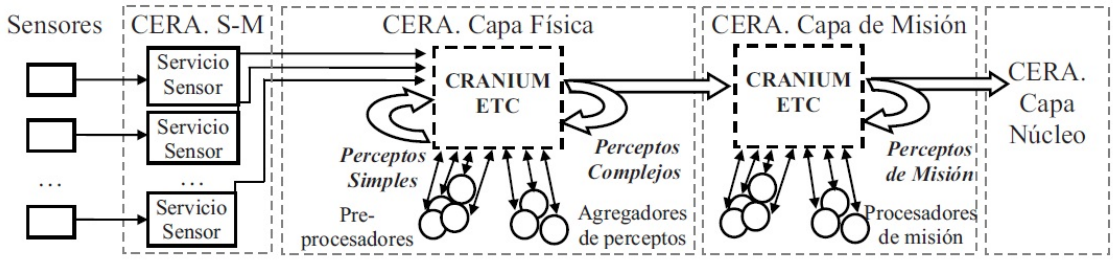

Figura 3.3: CERA-CRANIUM flujo bottom-up: percepción (extraído de [18])

Además del **flujo bottom-up** de los procesos de percepción, existe simultáneamente un **flujo top-down** en los mismos ETC para generar las acciones del bot. La capa física y la capa de misión específica incluyen acciones simples (traducciones directas de comandos Pogamut), comportamientos simples y comportamientos de misión 3.4

Una de las diferencias clave entre los flujos bottom-up y top-down de CERA-CRANIUM es que mientras los perceptos se componen iterativamente para obtener otros más complejos y representaciones con mayor significado, los comportamientos de alto nivel son fragmentados iterativamente hasta obtener una secuencia de acciones atómicas.

CERA-CRANIUM selecciona la siguiente acción de entre las que podrían ser potencialmente ejecutadas de manera, sí y sólo sí **se encuentra cerca del contexto activo** (contexto actualizado periódicamente por la capa núcleo de CERA).

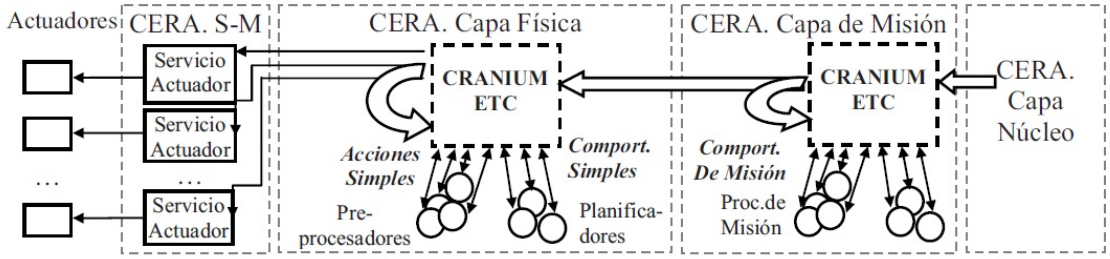

Figura 3.4: CERA-CRANIUM flujo top-down: generación de comportamiento (extraído de [18])

Generación de comportamiento

Tener un ETC donde convergen los flujos de información sensorial y motora facilita la implementación de múltiples bucles de realimentación para un comportamiento adaptado y efectivo. El comportamiento simple ganador está continuamente enfrentado a nuevas opciones generadas en la capa física, y por consiguiente existe un mecanismo para interrumpir comportamientos en progreso tan pronto como no se consideren la mejor opción. En términos generales, los procesos de activación o inhibición de la percepción y generación de comportamientos están modulados por CERA de acuerdo con el modelo cognitivo de consciencia implementado. En otras palabras, los comportamientos son asignados a un nivel de activación en función de su distancia con el contexto activo en términos del espacio sensoriomotor disponible. Sólo la acción más activa es la que se ejecuta al final de cada "ciclo cognitivo".

La distancia a un contexto dado se basa en criterios sensoriales tales como la localización relativa o el tiempo. Por ejemplo, si tenemos dos acciones: Acción A: "dispara a la izquierda" y Acción B: "dispara a la derecha", y un contexto activo apuntando al lado izquierdo del bot (porque hay un enemigo allí), la acción A será elegida con menor probabilidad para la ejecución, y la acción B será elegida en la cola de ejecución (siempre y cuando no sea muy antigua).

La figura 3.5 muestra una representación esquemática de los típicos bucles de realimentación producidos en la arquitectura CERA. Estos bucles se cierran cuando el bot percibe las consecuencias de las acciones ejecutadas, desencadenando respuestas adaptativas a diferentes niveles.

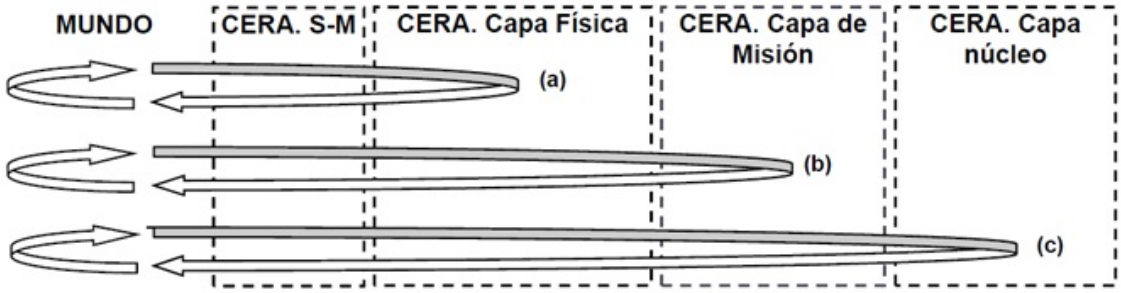

Figura 3.5: Diferentes bucles de realimentación producidos en CERA-CRANIUM (extraído de [18])

- La **curva (a)** en la figura 6 representa el bucle de realimentación que se produce cuando se desencadena un reflejo instintivo.
- La **curva (b)** corresponde a una situación en la cual el comportamiento específico de misión se realiza inconscientemente.

- Finalmente, la **curva (c)** ilustra el bucle de control de alto nivel, en el cual una tarea ha sido ejecutada conscientemente.

Estos tres tipos de bucles de control no se encuentran en exclusión mutua; de hecho, los mismos perceptos contribuirán habitualmente a diferentes bucles que tendrán lugar a diferentes niveles.

En esta implementación, los procesadores especializados se crean en el propio código, y se asignan dinámicamente a la correspondiente capa CERA. El Workspace correspondiente a cada capa será el encargado de gestionar la información proveniente de cada uno de estos procesadores y de la comunicación con el resto del sistema.

Counscious-Robots bot en acción

Lo que viene a continuación es un fragmento de un **típico flujo de perceptos** que en última instancia genera el comportamiento del bot (figura 7):

1. El procesador EnemyDetector detecta un nuevo enemigo, y crea un nuevo percepto "enemigo detectado".
2. El percepto "enemigo detectado" se recibe ordenadamente en el procesador SelecEnemyToShoot, el cual se encarga de seleccionar el enemigo al que vamos a disparar. Cuando se elige el enemigo, se genera la correspondiente acción de disparo.
3. Dos procesadores reciben la acción de disparar, una a cargo de apuntar al enemigo y disparar y la otra que crea nuevas acciones de movimiento para evitar el fuego enemigo.

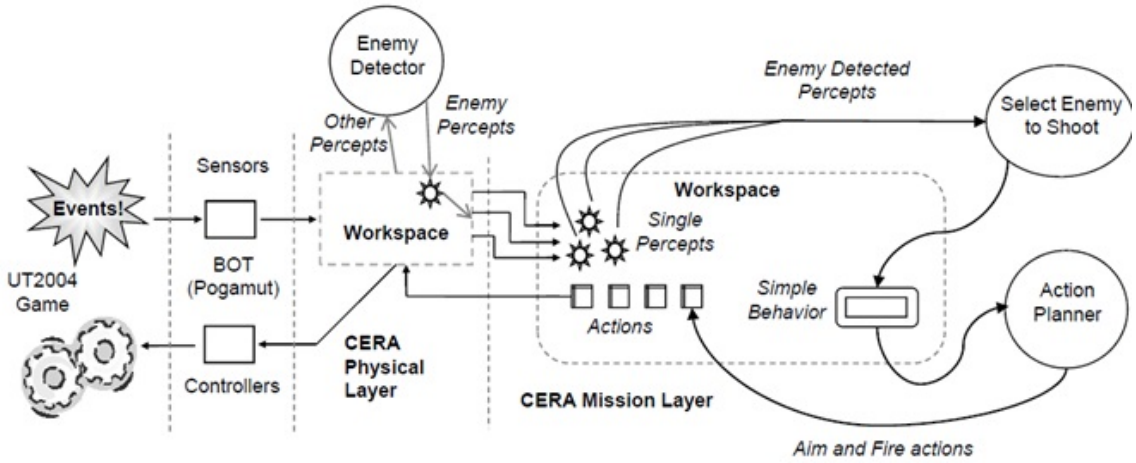

Figura 3.6: Esquema simplificado del flujo de perceptos y acciones (extraído de [18])

Este es un ejemplo muy simple de como funciona el bot. Sin embargo, lo habitual es tener escenarios más complejos en los cuales multitud de enemigos estén atacando al bot simultáneamente y el objetivo seleccionado podría ser cualquiera de ellos. En estos casos, los mecanismos de atención juegan un papel clave.

El diagrama de la Figura 3.7 representa de forma esquemática y más en profundidad la estructura global de la arquitectura CERA-CRANIUM y más en concreto de la implementación del CC-Bot2.

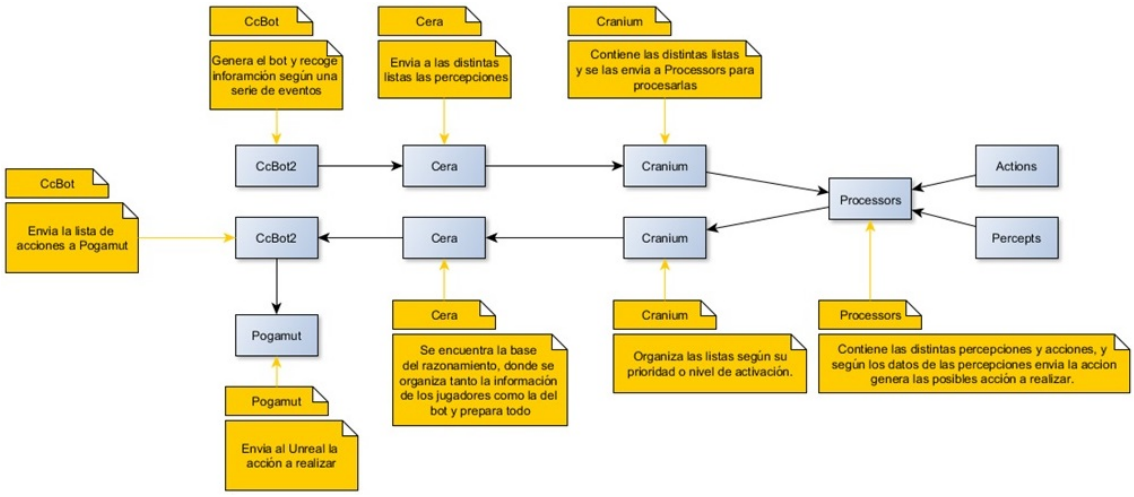

Figura 3.7: Diagrama general de CCBot2

3.3. Evaluación de CERA-CRANIUM y CCBot2

En esta sección evaluaremos el modelo CERA-CRANIUM en función de la definición de arquitectura cognitiva construida en el Capítulo 3 y también el nivel ConScale (Ver anexo) de la implementación CCBot2. Estas evaluaciones nos ayudarán a definir una hoja de ruta para elegir qué mejoras debemos añadir tanto a la arquitectura en general como específicamente al bot.

3.3.1. Evaluación de CERA-CRANIUM como arquitectura cognitiva

Puesto que la arquitectura CERA-CRANIUM está basada en la Teoría del Espacio Global de Trabajo de Baars y no presenta diferencias significativas de funcionamiento con respecto a la arquitectura Global Workspace de Shanaman [37, 38], asumimos que las características de CERA-CRANIUM se corresponden con las de Global Workspace a efectos del análisis de la tabla anterior.

Como se puede observar, la **única característica que no aparece referenciada** en CERA-CRANIUM es la de **Adaptación**, por lo que ser requiere de manera casi obligatoria la integración de algún mecanismo adaptativo. Es importante que esta mejora se realize en el conjunto global de la arquitectura y no en una implementación concreta para que la posterior evolución de estas implementaciones no se vea sujeta a las limitaciones de dicha implementación y la explotación de CERA-CRANIUM aplicada a distintos ámbitos se pueda realizar de manera transparente.

3.3.2. Evaluación ConScale de CCBot2

En la siguiente figura se presentan los resultados obtenidos al aplicar el Proceso de Evaluación Simplificado (PSE) sobre la implementación CC-Bot2. El cuadro contiene en su primera columna la lista de habilidades que abarca este modelo y más abajo se indica el valor del CQS, en la segunda columna se representan los perfiles cognitivos del modelo y finalmente, en la tercera columna, se muestra el nivel conceptual de ConsScale alcanzado. Es conveniente resaltar que el PSE simplemente proporciona una aproximación de lo que podría ser el nivel real ConsScale de una implementación.

CC-Bot2 cumple con algunas características de los niveles 3, 4 y 5. Sin embargo, se clasifica como un **agente de nivel 2** porque ConScale requiere el cumplimiento completo de los niveles

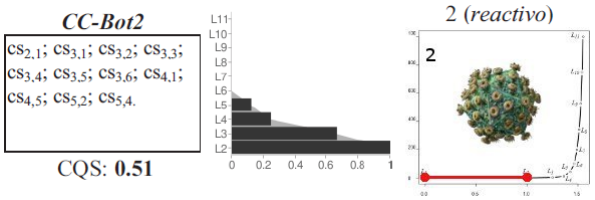

Figura 3.8: Evaluación PSE del CCBot2

inferiores para poder ser calificado como perteneciente a un determinado nivel i. El índice CQS de un agente reactivo puro es 0.18. Sin embargo, la puntuación de CCBot2 (0.51) indica que el agente presenta capacidades cognitivas adicionales (como se puede observar en el perfil cognitivo asociado que aparece en la Tabla 13). Asimismo, CC-Bot está lejos de alcanzar el nivel 4, para el cual el CQS asociado sería de 12.21 o superior.

En la Tabla 3.1 se enumera la lista de habilidades cognitivas ya implementadas en el modelo CCBot-2.

CS	Requisito
2,1	Capacidad de producir respuestas motoras fijas o reactivas (reflejos).
3,1	Adquisición autónoma y adaptativa de nuevas respuestas reactivas.
3,2	Uso de los sensores propioceptivos para generar respuestas adaptativas.
3,3	Selección de información sensorial relevante.
3,4	Selección de información motora relevante.
3,5	Selección de información relevante en memoria.
3,6	Evaluación (positiva o negativa) de objetos o sucesos seleccionados.
4,1	Aprendizaje por prueba y error. Re-evaluación de los sucesos seleccionados.
4,5	Representación espacial relativa (depictive) de los objetos percibidos.
5,2	Persecución de múltiples metas.
5,4	Aprendizaje por refuerzo autónomo.

Tabla 3.1: Requisitos ConScale cumplidos

Nuestro objetivo es, de manera casi inmediata, implementar la habilidad cognitiva restante para poder **completar el nivel 3** y, una vez superada esta barrera, analizar los **diferentes requisitos de los niveles 4, 5 y** (en menor medida) **6, para implementar algunos de ellos** y conseguir el mayor avance posible en la escala. En la Tabla 3.2 se recogen todo estos requisitos.

3.3.3. Evaluación global y trabajo a realizar

Es esencial que la arquitectura CERA-CRANIUM posea la característica de **Adaptación**. La opción más viable para implementar esta opción es crear una arquitectura híbrida basada en CERA-CRANIUM e integrar un módulo de comportamiento cognitivista. De esta manera se mantienen todas las propiedades de la arquitectura base y la propia estructura de CERA-CRANIUM nos permite realizar una integración casi modular de este nueva característica.

La integración de esta característica en el sistema **facilitará además el desarrollo de algunos de los requisitos de los niveles 4, 5 y 6** de ConScale para la nueva implementación. El resto de requisitos han sido analizados y de entre ellos se han elegido los que *a priori* complementarían esta característica de adaptación y no requerirían una nueva reestructuración de la arquitectura cognitiva.

CS	Requisito
3,7	Selección de los contenidos que necesitan ser almacenados en memoria.
4,2	Comportamiento dirigido hacia determinados objetivos, como seguimiento y escape o acercamiento y alejamiento de enemigos.
4,3	Evaluación del propio rendimiento en la consecución de una meta simple, descartando las acciones que no contribuyen a las metas perseguidas.
4,4	Capacidad básica de planificación: cálculo de las n siguientes acciones secuenciales.
5,1	Evaluación del rendimiento en la consecución de múltiples metas.
5,3	Habilidad para cambiar de contexto entre múltiples tareas.
5,5	Capacidad avanzada de planificación teniendo en cuenta todas las metas activas.
5,6	Capacidad para generar contenidos mentales seleccionados con significados basados en la interacción del agente con el medio (grounded meaning).
6,1	Evaluación del estado propio (emociones globales).
6,2	Las emociones globales causan efectos en el cuerpo del agente.
6,3	Representación de los efectos de las emociones en el organismo (sentimientos).
6,4	Capacidad para mantener un mapa preciso y actualizado del esquema corporal.
6,5	Aprendizaje abstracto (generalización de lecciones aprendidas).
6,6	Capacidad para representar un flujo de perceptos integrados que incluyan el estado propio.

Table 3.2: Requisitos ConScale Implementables

Capítulo 4

Descripción del nuevo modelo: CCBotSoar

En este capítulo se detallan las mejoras realizadas en la arquitectura para desarrollar la característica de adaptación (el Sistema de Toma de Decisiones) y las nuevas implementaciones complementarias como son la Memoria de Objetos y la simulación de emoción "Riesgo".

4.1. Adaptación

De entre las cuatro arquitecturas cognitivas analizadas en el Capítulo 2 que presentan la característica de adaptación, Soar y ACT-R son las más utilizadas y representativas del paradigma y también las más sencillas de integrar en un sistema híbrido. A la hora de elegir una de ellas como herramienta para el desarrollo de esta nueva característica:

- Ambas tienen capacidades deliberativas y de aprendizaje y por lo tanto cumplen las características deseadas, posibilitan la representación del estado del bot, tienen mecanismos para la comunicación con un entorno externo a la arquitectura cognitiva que será el videojuego Unreal Tournament 2004 y disponen de diferentes métodos de aprendizaje.
- Como se va a utilizar Pogamut para la implementación del bot es necesario que la arquitectura seleccionada disponga de una versión en Java para facilitar así la conexión entre Pogamut y la arquitectura. Ambas disponen de versión en java, **jSoar**[1] y **jACT-R**[2], luego esto no nos obliga a utilizar una arquitectura concreta.
- ACT-R presenta frente a Soar la ventaja de ser híbrida, no obstante, su mecanismo de aprendizaje de nuevas producciones no es tan potente como el *chunking* de Soar. Soar ademas dispone de un mecanismo integrado de aprendizaje por refuerzo. Como el bot esta centrado en la capacidad de aprender de sus propias experiencias este hecho diferencial entre Soar y ACT-R es el que decanta la balanza hacia Soar.

Soar es la arquitectura seleccionada para el desarrollo del sistema de toma de decisiones del bot. Para más información sobre Soar se puede consultar el Anexo D y el manual de Soar disponible con cada distribución de Soar[3].

[1] http://code.google.com/p/jsoar/
[2] http://jactr.org/
[3] http://code.google.com/p/soar/wiki/SoarTutorial

4.1.1. Soar

Soar (*State, Operator And Result*) es una arquitectura cognitiva basada en la teoría de la mente propuesta por Newell[2], para el desarrollo de sistemas que exhiben un comportamiento inteligente. Es definida por sus creadores como una arquitectura de propósito general en el sentido de que Soar no tiene intención de tratar únicamente el problema de la inteligencia humana sino de tratar el problema de la inteligencia en general. El proyecto Soar comenzó en la Universidad Carnegie Mellon por Newell, Laird y Rosenbloom como un test para las teorías de la cognición de Newell[2].

Su diseño esta basado en la hipótesis de que toda conducta orientada a unos objetivos, puede ser tratada como la selección y aplicación de unos operadores sobre un estado:

1. el estado es la representación de la situación actual del problema a resolver,
2. los operadores transforman el estado haciendo cambios en la situación actual y
3. el objetivo es la situación que se desea alcanzar durante la resolución del problema.

Soar consta fundamentalmente de un interfaz de entrada/salida que interactua en cada ciclo de ejecución y posee tres tipos de memoria.

- **Memoria de trabajo:** Esta compuesta por los *Working Memory Elements* (WME) que son tuplas "identificador-atributo-valor" que contienen la representación del estado actual. Permiten la organización jerárquica y atributos con múltiples valores. Se ve afectada por las decisiones del ciclo de ejecución y por la interfaz de entrada/salida.
- **Memoria de producción:** Contiene el denominado conocimiento a largo plazo compuesto por producciones (es decir, reglas del tipo "if→then" o "condición -> acción"). En cada ciclo se disparan las producciones cuyas condiciones coinciden con el estado representado en la memoria de trabajo.
- **Memoria de preferencias:** Guarda las preferencias asignadas a los operadores, que determinan su elección durante el ciclo de ejecución.
- **Interfaz de entrada/salida:** Es la conexión de Soar con el entorno externo en el cual se ejecuta. El interfaz de entrada actualiza al comienzo de cada ciclo de ejecución los WME que Soar tiene en la memoria de trabajo y el interfaz de salida al final de cada ciclo se encarga de transmitir al entorno la acción a realizar.

El funcionamiento de Soar esta basado en una secuencia de acciones llamada *execution cycle* (ciclo de ejecución) que se ejecuta continuamente. Este ciclo aplica el operador vigente y selecciona el próximo operador a aplicar hasta que la meta sea alcanzada (Figura 4.1). En cada ciclo de ejecución se selecciona un único operador.

Si en un ciclo de ejecución no es posible resolver la selección del operador Soar alcanza un *impasse*, que genera un nuevo subestado para la resolución del impasse. Se produce así una descomposición de objetivos. Cuando los sucesivos subestados se van resolviendo, la arquitectura almacena todo el proceso en una nueva producción. Cuando se resuelve un impasse, Soar "aprende" ya que almacena el conocimiento sobre como resolver el impasse en una producción. Éste es el principal mecanismo de aprendizaje en Soar, llamado *chunking*. La nueva producción, llamada *chunk*, tiene como condiciones los elementos del estado que generaron el impasse y como acción los cambios en la memoria de trabajo o en las preferencias que resolvieron el impasse, y es almacenado en la memoria de producción con el resto de producciones.

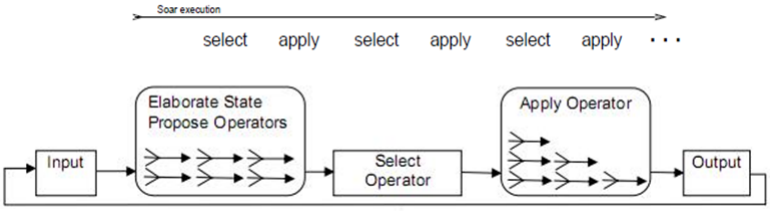

Figura 4.1: Ciclo de ejecución de Soar (extraído de [2])

Soar dispone de un sistema de **aprendizaje por refuerzo** que modifica el conocimiento sobre la selección de operadores mediante un sistema de recompensas. Este sistema de aprendizaje por refuerzo esta integrado con la memoria de producciones. Utiliza las preferencias numéricas que son modificadas mediante las recompensas en función de nuestros intereses para así aprender qué operador es más conveniente seleccionar en un estado u otro.

Para más información sobre **Soar** se puede consultar el Anexo D y el manual de Soar disponible con cada distribución.

4.2. Memoria de objetos

Durante el transcurso de una partida de Unreal Tournament 2004 (y en la de casi cualquier otro juego FPS) la posición de los objetos (munición, salud y otros objetos especiales) sea siempre la misma. Los jugadores habitualmente **recuerdan la posición de estos objetos** ya sea por que éstos se encuentran en puntos estratégicos muy fáciles de identificar (y más cuando la mayoría de los mapas tienen cierta simetría) o porque por el propio desarrollo de la partida, consiguen acordarse.

Implementar este comportamiento en nuestro bot mejorado es tan sencillo como crear un nuevo procesador de perceptos.

Este nuevo procesador (bautizado como RememberItems en la actual implementación) captura los perceptos de "objeto recogido" y almacena sus posiciones en dos listas diferentes:

- una para todos los **objetos que incrementan o recuperan nuestra salud**, así como los distintos tipos de **escudos de energía**
- y otra para el **resto de objetos** (principalmente **municiones** y **armas**, aunque también se incluyen potenciadores de disparo)

El procesador gestiona y actualiza estas listas con cada nuevo objeto recogido **almacenando la posición del objeto** aunque no su tipo, pues en algunos casos el juego crea puntos en los que los objetos que se pueden recoger cambian a lo largo del tiempo y tampoco sería realista que un jugador recordase exactamente el tipo de objeto concreto que aparece en un punto.

El mismo procesador puede capturar perceptos sobre el nivel de salud y munición, de manera que cuando estos niveles **se encuentran por debajo de un mínimo** (definidos globalmente y utilizados por, por ejemplo, el procesador que nos ayuda a huir de los enemigos) propondrá al sistema una **acción de movimiento dirigida hacia el objeto más cercano** que se corresponda con su necesidad (salud o munición).

El siguiente diagrama representa gráficamente el funcionamiento interno de este procesador.

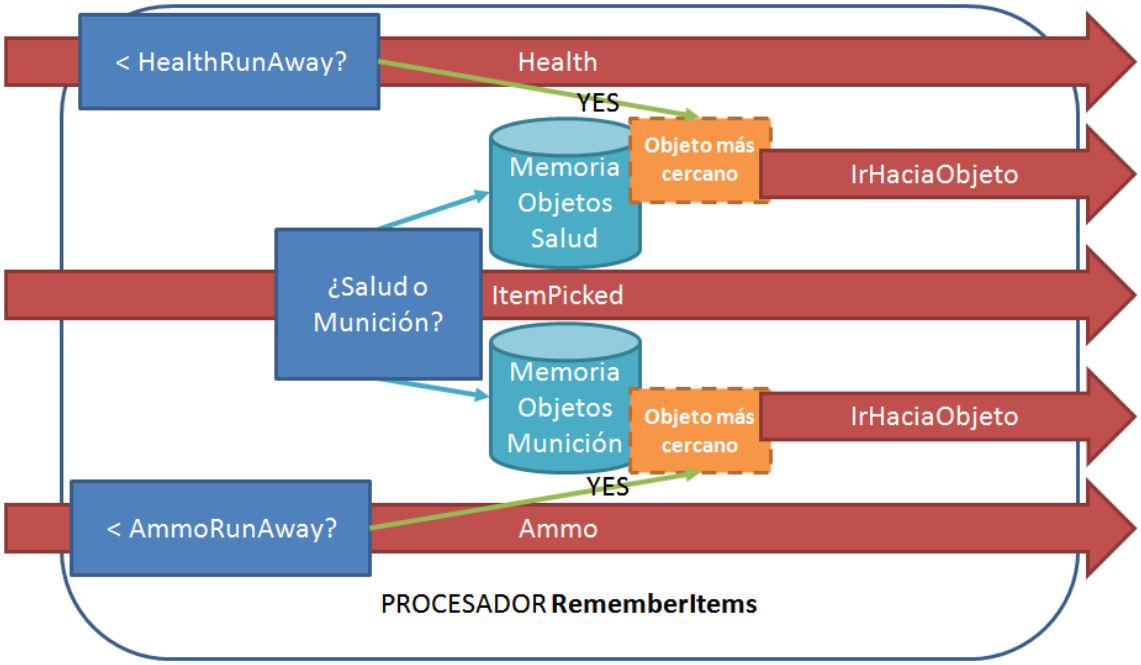

Figura 4.2: Procesador RememberItems

4.3. Sistema de toma de decisiones

Como extensión a la nueva característica de **Adaptación** y como parte de la creación de una memoria a largo plazo se ha implementado un sistema de toma de decisiones centrado en la capacidad de aprender de sus propias experiencias e implementado "emociones" en su comportamiento. En particular, el sistema ha sido desarrollado en base a los conceptos de motivación, *drive*, emoción y aprendizaje por refuerzo. Los conceptos de motivación y *drive* forman parte de la representación del estado bot, más concretamente del estado interno del bot.

Este sistema de ha realizado utilizando como inspiración el sistema de toma de decisiones desarrollado en [39], añadiendo las modificaciones necesarias para su funcionamiento como módulo anejo a la estructura de CERA-CRANIUM, facilitando así su integración en la arquitectura.

4.3.1. Aprendizaje por refuerzo

El bot y el entorno interactúan continuamente, el bot selecciona acciones y el entorno responde a dichas acciones y presenta nuevos estados para el bot. A cada instante, el bot calcula una función desde los estados a cada una de las acciones. A esta función se le llama política de aprendizaje. Los métodos de aprendizaje por refuerzo especifican cómo el agente cambia su comportamiento como resultado de su experiencia. El objetivo del bot es maximizar la cantidad de recompensas que recibe a lo largo del tiempo [8].

La valoración de los comportamientos cuando el bot está en un cierto estado, se realiza utilizando algoritmos de aprendizaje por refuerzo. El bot aprenderá qué acción escoger cuando se encuentra en un estado determinado. El refuerzo utilizado para valorar el resultado de una acción es el *appraisal*, definido en función de la variación que experimenta el bot en su *mood*. El *mood* es una función de las necesidades del agente, por lo que este refuerzo mide el efecto de la acción realizada en las necesidades del bot. Las variaciones positivas y negativas del *mood* están inspiradas en

las emociones de felicidad y tristeza respectivamente. El bot, por lo tanto, utiliza estas emociones para valorar sus propias acciones y aprender cuáles son las más apropiadas en cada estado.

A pesar de haberse convertido en uno de los algoritmos de aprendizaje más usados, Q-learning no sirve en nuestro entorno, ya que el videojuego Unreal Tournament es un entorno muy caótico que cambia muy rápido, entonces en el momento de realizar la política de Q-learning la mejor opción del estado futuro puede tener un Q-valor muy elevado pero en realidad ese estado no llega a producirse porque ha cambiado el entorno muy rápido.

En su lugar utilizamos el algoritmo denominado **Sarsa** [11],

$$\delta_t = \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

donde:

- $Q(s_t, a_t)$ es el Q-valor de la pareja estado-acción en el ciclo t.
- $Q(s_{t+1}, a_{t+1})$es el Q-valor de la pareja estado-acción elegidos en el siguiente ciclo de decisión.
- r_{t+1} es la recompensa total obtenido durante el siguiente ciclo de ejecución.
- α es la tasa de aprendizaje, controla cuanta importancia se le da a la recompensa más reciente.
- γ es el factor de descuento, define cuanto afectan las recompensas futuras.

cuya diferencia con Q-learning radica en que para Sarsa los valores se actualizan en el siguiente ciclo de ejecución considernado el estado al que el sistema ha llegado y, en ese, eligiendo la mejor opción. Debido a las características caóticas del entorno se ha elegido Sarsa como algoritmo de aprendizaje por refuerzo.

4.3.2. Gestión de emociones

Las emociones son tratadas en base a la teoría de Schachter y Singer [13], que propone que las emociones son una entidad producto de dos aspectos: alteración personal y valoración cognitiva. Los fenómenos emocionales están formados por una respuesta genérica del sistema autónomo ("*arraisal*") y por la evaluación cognitiva de tal alteración ("*appraisal*"). Las emociones son interpretaciones cognitivas de ciertas situaciones.

Magna Arnold [12] entiende que las emociones son la combinación de ambos factores:

1. la valoración mental del daño o beneficio potencial de una situación, y
2. la tendencia a la acción que nos acerca a las situaciones evaluadas positivamente o que nos aleja de las situaciones no deseables evaluadas negativamente.

La toma de decisiones será aprendida a través de la propia experiencia del bot (sus "éxitos" y sus "fracasos"). A través de ellos, mediante el aprendizaje por refuerzo, el bot aprende las políticas de comportamiento adecuadas. Las emociones asociadas a estos procesos se inspiran en emociones básicas como el "*mood*".

Como función de refuerzo se utiliza el ***appraisal*** que está relacionado con la variación del *mood* del bot.

Los ***drives*** o necesidades del bot varían su valor en cada paso de ejecución siguiendo cada *drive* una dinámica determinada. Dicho conjunto de dinámicas intenta dotar de "personalidad" al bot y, la variación del conjunto generará bots con diferentes personalidades. Estos valores del *drive*, junto con los valores de los estímulos externos sirven para calcular la intensidad de las motivaciones relacionadas con cada *drive*. La motivación de mayor intensidad es la motivación dominante y va a ser la que determine el estado interno del bot. Este estado interno junto con el estado externo, determina el estado global del bot.

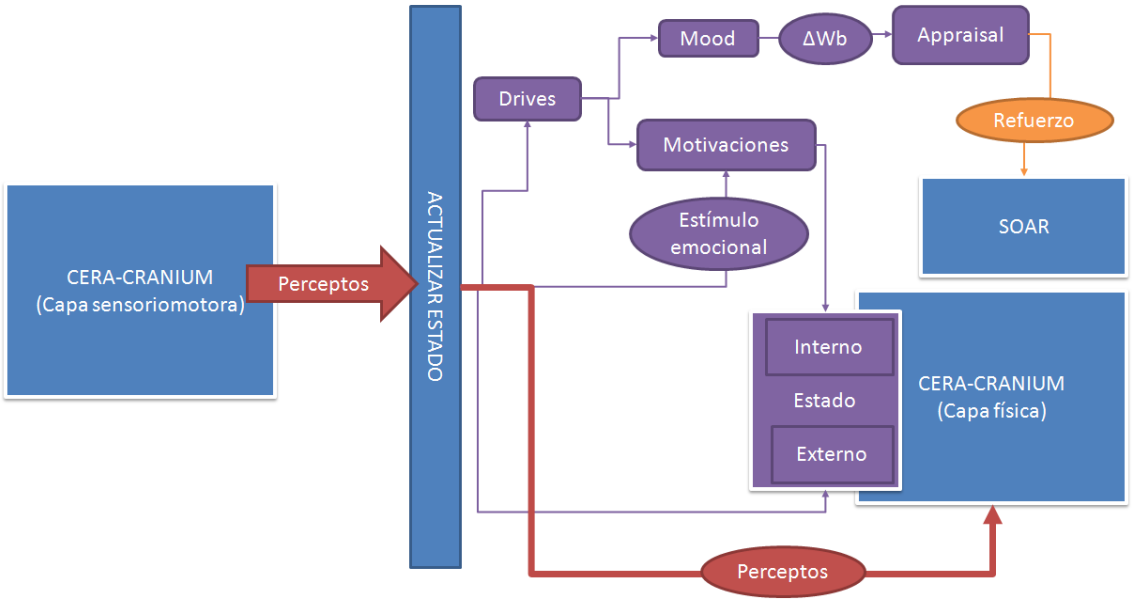

Figura 4.3: Sistema de toma de decisiones.

4.3.3. Estado del bot

Es necesario conocer el estado del bot para el proceso de toma de decisiones. En este sistema el estado del agente es la combinación de su estado interno y su estado externo. $S = S_{interno} \times S_{externo}$. El estado interno del agente depende de las motivaciones, que están ligadas a las necesidades, es decir, a los *drives* del bot.

Estado interno

El **estado interno** está compuesto por la motivación dominante y los niveles de salud y munición. Los posibles valores del estado interno están reflejados en la tabla 4.1. Se considera que la salud es alta cuando es superior a 100, baja cuando es inferior a 40 y media en el resto de casos. La munición se considera alta cuando es superior a 100, baja cuando es inferior a 50 y media en el resto de casos.

Estado interno	Valores
Motivación dominante	Agresividad, Enfermedad, Incapacidad, Ok
Salud	Alta, Media, Baja
Munición	Alta, Media, Baja

Cuadro 4.1: Estado interno.

Los ***drives*** del bot están relacionados con necesidades fisiológicas y con necesidades de interacción social y seguridad. A medida que esta necesidad aumenta, lo hace la intensidad del *drive*. El valor inicial e ideal de todos los *drives* es cero. En este caso se considera que un *drive* esta satisfecho pues no existe necesidad.

Los drives que utiliza nuestro sistema están pensados para simular las necesidades que tiene un jugador de Unreal Tournament 2004 (y por extensión a la mayoría de FPS) y que rigen su comportamiento. Se busca que dicho comportamiento bot sea similar al de los jugadores humanos. Los *drives* considerados en este sistema son los siguientes:

- **Agresividad**: En el ámbito de la interacción social, simula la tendencia a matar enemigos que tiene todo jugador Unreal, ya que esto es la base del juego y quién más enemigos mate obtiene la victoria.
- **Enfermedad**: En el ámbito de las necesidades fisiológicas, simula la necesidad de obtener salud para evitar la muerte.
- **Incapacidad**: En el ámbito de la necesidad de seguridad, este indicador simula la necesidad de obtener y poseer munición para poder matar a los enemigos y que estos no te maten a ti.

Los *drives* agresividad, enfermedad e incapacidad se incrementan una cierta cantidad a cada paso de ejecución, pero no aumentan al mismo ritmo. Los *drives* enfermedad e incapacidad también están ligados a los valores de salud y munición del bot incrementando su necesidad cuando estos valores son bajos y disminuyéndola cuando tenemos mucha salud o mucha munición.

La tabla 4.2 resume los efectos que tienen determinadas acciones o acontecimientos sobre el valor del *drive*.

Acción	**Drive**	**Efecto**
Coger botiquín	Enfermedad	Satisfacción del drive (reducir a cero)
Coger munición	Incapacidad	Satisfacción del drive (reducir a cero)
Eliminar enemigo	Agresividad	Satisfacción del drive (reducir a cero)
Que te disparen	Enfermedad	Aumento dependiente del nivel de salud
Disparar	Incapacidad	Aumento dependiente del nivel de munición

Cuadro 4.2: Efectos de las acciones en los drives.

La intensidad del drive interior se ve modificada por la fuerza del **estímulo externo**. Si el valor del drive es bajo, se necesita un estímulo fuerte para activar el comportamiento motivado. Por otro lado, si el drive es alto, un estímulo medio será suficiente[7].

Siguiendo esta idea, la intensidad de las motivaciones en la arquitectura (M_i) es la suma de la intensidad del drive relacionado (D_i) y el valor del estímulo externo relacionado (w_i), tal y como se expresa en la siguiente ecuación:

$$M_i = D_i + w_i$$

Los estímulos externos o motivacionales son determinados objetos que el bot encuentra por el mundo (la tabla 4.3 contiene cual es el estímulo de cada motivación). De acuerdo con la ecuación de la intensidad de la motivación, una motivación puede tener una intensidad alta por dos motivos:el valor del *drive* correspondiente es alto o el estímulo motivacional asociado está presente.

Este modelo puede explicar el hecho de que, ante la disponibilidad de comida delante de nosotros, algunas veces comamos sin tener mucho hambre.

Motivación	**Estímulo**
Enfermedad	Botiquín de salud
Incapacidad	Cartucho de munición
Agresividad	Enemigos

Cuadro 4.3: Estímulos motivacionales

Una vez se calculan las intensidad de todas las motivaciones, éstas entran en competición. La motivación con la intensidad más alta es la motivación dominante y determina el estado interno del agente. Existe un nivel límite de activación (L_d) que tiene que ser superado por el drive de la motivación para que ésta sea elegida como motivación dominante.

$$\text{Si } D_i < L_d \Rightarrow M_i = 0$$
$$\text{Si } D_i > L_d \Rightarrow \text{se aplica la ecuación.}$$

Mientras todos los *drives* tengan un valor inferior a L_d se considera que el bot no tiene ninguna necesidad, o no hay motivación dominante, y el bot se encuentra en un estado ideal en el que todas sus necesidades están satisfechas (Ok).

Estado externo

El **estado externo** es el estado del bot en relación a algunos elementos, pasivos y activos, del entorno Unreal Tournament que puedan interactuar con el bot. Se ha simplificado el estado externo porque las posibilidades son inmensas y de esta manera se evita que haya un número muy elevado de estados del bot en relación a todos los objetos. La tabla 4.4 contiene los elementos que componen el estado externo.

Estado Externo	**Valores**
Veo Salud	Sí, No
Veo Munición	Sí, No
Me Atacan	Sí, No
Veo Enemigo	Sí, No
Peligro	Nulo, Bajo, Medio, Alto, Mortal

Cuadro 4.4: Estado Externo

Función de refuerzo

Los *drives* del bot están relacionados con sus necesidades. Se define el ***mood*** del bot como el grado de satisfacción de sus necesidades, de manera que, cuando todos los *drives* del bot están satisfechos el *mood* es máximo. El *mood* del bot es una función de los valores de sus *drives* y factores de personalidad α_i que valoran la importancia de cada *drive* con respecto al *mood* del bot, siendo

$$Wb = Wb_{ideal} - \sum \alpha_i * D_i$$

Denotamos Wb_{ideal} el valor ideal de *mood* del bot. A medida que aumentan los valores de los *drives,* el *mood* del agente disminuye. Dependiendo de los valores de los factores de personalidad, el aumento de los *drives* puede afectar en mayor o menor medida al *mood*. El valor del *mood* se calcula en cada paso de simulación, además de su variación ($\triangle Wb$). Este incremento se puede calcular como

$$\triangle Wb^{k+1} = Wb^{k+1} - Wb^{k}$$

Rolls, en [9] propone que las emociones son estados provocados por refuerzos (recompensa o castigo), de manera que nuestras acciones estarán dirigidas a obtener recompensas y evitar castigos. Siguiendo este punto de vista, en el sistema de toma de decisiones se va a utilizar el ***appraisal*** como refuerzo positivo y negativo dentro del sistema de aprendizaje por refuerzo. De esta forma el refuerzo también esta relacionado con la reducción de los *drives*, ya que las variaciones positivas del *mood* implican una reducción de los *drives* y las negativas un aumento.

Como se ha comentado, el ***appraisal*** es la evaluación cognitiva de la alteración personal que produce una emoción. En nuestro sistema se define en función de la variación que experimenta el *mood*:

$$\text{Si } \triangle Wb > Lh \Rightarrow Appraisal \text{ positivo}$$
$$\text{Si } \triangle Wb < Ls \Rightarrow Appraisal \text{ negativo}$$

donde $\triangle Wb$ es la variación del *mood* y $Lh \geq 0$ y $Ls \leq 0$ son las variaciones mínimas del *mood* del agente que producen *appraisal* negativos o positivos.

4.4. Mapa de Peligro

Habitualmente, los jugadores conocen casi a la perfección los llamados "puntos calientes" de los mapas multijugador, es decir, las **zonas del escenario más peligrosas** (donde es más probable matar y ser matado). Este peligro puede ser consecuencia de que la zona se encuentre en la línea de fuego de una posición oculta al jugador desde la que el enemigo pueda dispararnos con facilidad, por ser una zona de tránsito casi obligatorio del mapa y por tanto se congreguen muchos enemigos, o simplemente porque sea fácil acabar siendo víctima de algún elemento del escenario (lava, arenas movedizas, el vacío, etc).

Por lo tanto, es interesante contar con un "mapa mental de peligro", análogamente al que "poseen" los jugadores fruto de su experiencia de juego o del propio conocimiento del terreno.

Cuando nuestro bot entra en una partida, cargará en su memoria los **datos geométricos** del mapa en el que se va a jugar. En concreto, se almacenan todos los puntos de navegación (*NavPoints* en el entorno Unreal Tournament 2004) definidos en el escenario y con ellos se crea una lista en la que se asignará un valor 0 a cada punto.

Cada vez que el bot sea aniquilado o vea como otro jugador muere, el valor correspodiente al punto de navegación más cercano al **lugar dónde sucedió la muerte se incrementará en un valor** η ya definido, mayor en el caso de la muerte propia pues suponemos que nuestra manera de jugar es habitualmente más segura frente a la de otros jugadores. Además de incrementarse el valor del punto más cercano, también **aumentará el valor de todos los puntos de navegación que se encuentren en un radio inferior o igual a un valor** τ en un cuarto del incremento ($\frac{\eta}{2}$) del punto central.

De esta manera estamos creando un mapa mucho menos preciso pero más realista con respecto las zonas del escenario que resulten peligrosas.

Cada vez que el bot se mueve se consulta este "mapa de peligro" informando de **cuál es el punto de navegación más cercano y obteniendo el valor *Peligro*** para nuestro estado interno. Dado que este valor numérico es decimal y su utilización conllevaría la creación de multitud de estados diferentes, se ha optado por asignar un valor verbal a cada valor numérico (siempre redondeando hacia abajo). La tabla (4.5) presenta la correspondencia de estos valores numéricos con su representación verbal.

Valor	**Peligro**
0	Nulo
1	Bajo
2	Medio
3	Alto
> 3	Mortal

Cuadro 4.5: Valores DeathMap

Aunque puede parecer que la densidad en la división de los valores inferiores a 3 así como dicho límite son muy pequeños, debemos recordar que la mayoría de los mapas multijugador de Unreal

Tournament 2004 son de gran tamaño, de manera que si se concentran más de tres muertes en el mismo punto concreto, significa que ciertamente es una zona a la que es peligroso acercarse.

En el caso de los mapas pequeños, la propia función de expansión de valor terminará tarde o temprano asignando el valor máximo de *Peligro* a todos los puntos de navegación del mapa, cumpliendo así la premisa de que en esta clase de mapas, a priori, no existen zonas más peligrosas que otras.

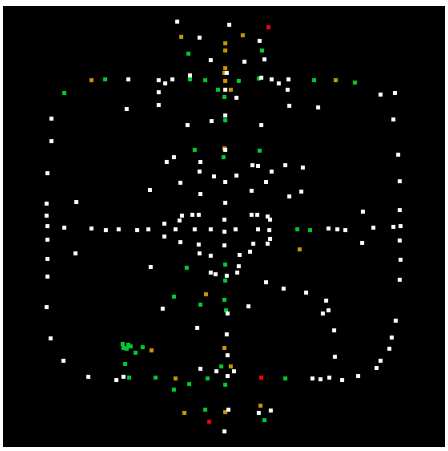

Figura 4.4: Ejemplo visual de mapa de peligro

4.5. Modelo de Riesgo

La simulación de emociones puede ser algo muy complejo si lo que se busca es imitar el fondo y no así la forma. En un juego de mecánica tan sencilla como el Unreal Tournament 2004, en el cual es prácticamente imposible que el comportamiento de un jugador refleje tristeza, alegría o melancolía, es, sin embargo, muy sencillo emular los sentimientos de un jugador humano de cara al resto de competidores.

Además de la simulación de emociones propuesta en el sistema de toma de decisiones, se propone la característica ***Riesgo*** como un añadido para enriquecerlo.

La propia experiencia como jugador me ha enseñado que en este tipo de juegos competitivos, la tendencia es que **la concentración se vaya perdiendo en función de nuestros resultados en la partida**. Esto sucede si por ejemplo nos han matado un montón de veces y nuestros enemigos parecen invencibles o si, por el contrario, no nos han matado ni una sola y nadie nos planta cara, nuestro comportamiento tiene a ser algo más errático y menos enfocado al juego, fruto de la euforia por una victoria casi segura o la resignación y pasividad provocada por una partida que claramente no sólo no vamos a ganar, sino que ni siquiera vamos a disfrutar.

Esta condición puede simularse simplemente **modificando el radio de selección de acciones** en el *workspace* de cada capa. El parámetro utilizado como criterio de selección es el nivel de activación de cada acción con respecto al contexto.

Dado que ambos extremos de la emoción provocan el mismo resultado, lo único que debemos hacer es crear un modificador (θ) a este radio de selección en función de nuestra posición en la partida. Calculando la diferencia relativa (en función de la puntuación necesaria para ganar la partida) con respecto a estos cuatro valores y asignando un peso a cada una de estas diferencias, obtenemos el modificador al radio de elección de acciones.

En nuestro bot, este modificador se calcula con la siguiente fórmula matemática basada en la clasificación de la partida:

$$\theta = 1 + \left| \frac{S_{own} - S_{max}}{S_{top}} \alpha_{max} + \frac{S_{own} - S_{min}}{S_{top}} \alpha_{min} + \frac{S_{own} - S_{next}}{S_{top}} \alpha_{next} + \frac{S_{own} - S_{prev}}{S_{top}} \alpha_{prev} \right|$$

donde:

- S_{own} es la **puntuación de nuestro bot**
- S_{top} es la **puntuación objetivo** de la partida
- S_{max} es la puntuación del jugador que **va ganando** la partida
- S_{next} es la del jugador que está s**ólo un puesto por encima** de nuestro bot
- S_{prev} es la puntuación del **siguiente jugador en la clasificación** de la partida
- S_{min} es la puntuación del **último clasificado** en la partida
- α_{max}, α_{min}, α_{next} y α_{prev} son los **diferentes pesos de estas diferencias** (la suma de los pesos debe ser 1) en función de la importancia que se le quiera dar a cada una de ellas.

El modificador θ se envía a la capa física y esta se encarga de almacenar la variable para uso de su workspace y de propagar el valor actualizado a las capas superiores.

4.6. Integración del sistema de toma de decisiones en CERA-CRANIUM

Como resultado del proyecto se integra el sistema de toma de decisiones en la arquitectura CERA-CRANIUM.

Para ello se utiliza el módulo **jSoar**, implementación en Java de una serie de funciones para cargar el fichero de reglas (las reglas de la memoria de producciones), ejecutar las fases del ciclo de ejecución e interactuar con el interfaz de entrada/salida.

El módulo de jSoar está integrado dentro de la arquitectura CERA-CRANIUM ya que es ésta quien maneja la ejecución del bot. La conexión se ha realizado mediante dos vectores, uno de entrada y otro de salida que se encuentran en el módulo de jSoar, y unas funciones para que CERA-CRANIUM pueda interactuar con ellos.

La figura 4.6 muestra un esquema general de la conexión.

En el funcionamiento original de CERA-CRANIUM cada ciclo de ejecución (alrededor de cuatro ciclos por segundo), la arquitectura seleccionaba una acción de cada tipo de entre las propuestas por el *workspace* en la capa física en función de su nivel de activación y ejecutaba todas.

Con la integración del sistema de toma de decisiones (y por ende de Soar), se modifica este funcionamiento de manera que cada vez que se ejecuta un ciclo, de entre todas las acciones propuestas se asigna a cada una un operador de Soar y se selecciona únicamente una haciendo uso del motor de reglas.

En Soar se encuentra el estado interno del bot y la capa física introduce en el vector de entrada toda la información que obtiene de los sensores del mundo externo, además de las acciones

Figura 4.5: Acciones antiguo

propuestas y el módulo de jSoar en su fase de Input del ciclo de ejecución lee la información de este vector de entrada y la introduce en su representación del estado.

De esta forma Soar dispone del estado del bot y por lo tanto, de las parejas "estado-acción" necesarias para el sistema de aprendizaje por refuerzo. Cada ciclo se actualizará el estado y se recompensará o penalizará la acción elegida. Para evitar que el módulo de Soar desarrolle un comportamiento paralelo a la arquitectura, se eliminarán de Soar las acciones propuestas que hayan sido finalmente elegidas.

En el vector de salida, el módulo de jSoar durante la fase Output del ciclo de ejecución escribe qué acción ha resultado seleccionada de entre las propuestas, y la capa física lee dicha acción del vector y la ejecuta. Las acciones no elegidas no se eliminan automáticamente, sino que aguardarán en la cola de propuestas hasta que la sean seleccionadas o hasta que la arquitectura decida que son demasiado antiguas.

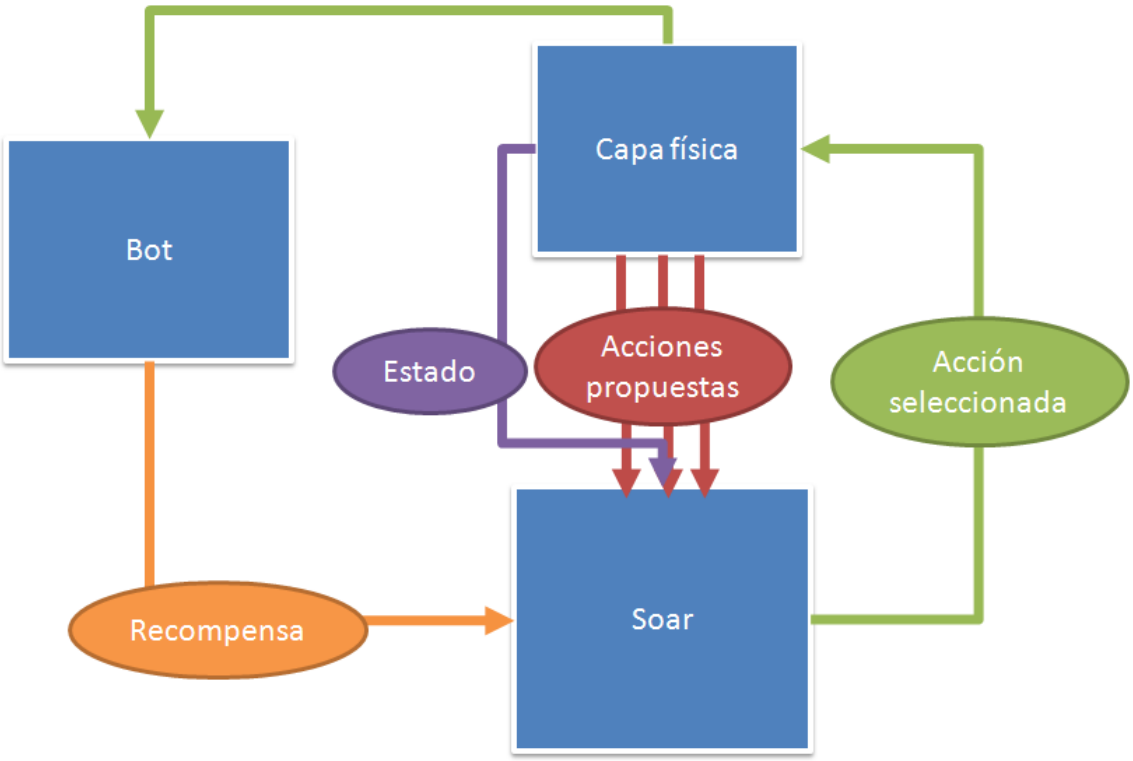

Figura 4.6: Esquema general de la conexión.

El mecanismo de aprendizaje por refuerzo de Soar requiere "resetearse" cada vez que se aplica para actualizar los Q-valores de las parejas estado-acción. En nuestro sistema el refuerzo es aplicado cada vez que hay una variación positiva o negativa del *mood* y cuando muere el bot. Se ha desarrollado una función que resetea jSoar para actualizar los Q-valores. Cada vez que haya que aplicar el refuerzo se le comunica a jSoar a través del vector de entrada indicando cual es la recompensa o penalización sobre el último par estado-acción ejecutado, entonces jSoar devolverá como acción "Reset" a través del vector de salida de la conexión para que pogamut ejecute la rutina de "reinicio" de jSoar y los Q-valores sean actualizados. De esta forma se integra el proceso de aprendizaje por refuerzo en la conexión de Pogamut y jSoar, y no supone más que añadir una rutina de "reinicio" de jSoar para que los valores sean actualizados.

Como ya se ha dicho, este proceso supone un ciclo de pogamut perdido durante la ejecución, que es utilizado para "reiniciar" el mecanismo de aprender, pero que no supone ningún problema ya que, durante este ciclo, el bot repetirá la última acción realizada, simulando así, además, el retraso en la velocidad a la hora de tomar decisiones y la propia inercia derivada de este.

Puesto que la memoria de producciones de Soar se reinicia cada vez que iniciamos una partida, el sistema dispone de un archivo en el que se van cargando todas las nuevas reglas creadas durante el transcurso de una partida para poder cargarlas en la siguiente. De esta manera simulamos una **memoria a largo plazo** que se irá enriqueciendo a cada partida que juegue nuestro bot, llegando a crear una suerte de personalidad propia en función de sus experiencias pasadas.

En las Figuras 4.8 y 4.7 se resume de manera gráfica la integración del Sistema de Toma de Decisiones (STD) en los flujos top-down y bottom-up de la arquitectura CERA-CRANIUM.

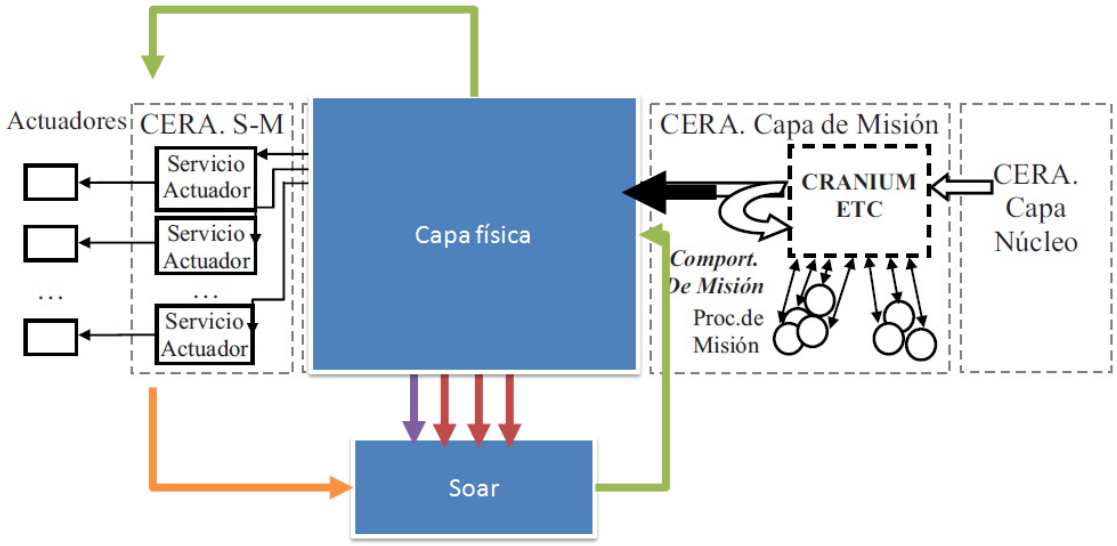

Figura 4.7: Integración del STD en el flujo bottom-up

4.7. Detalles sobre la implementación

Todas estas nuevas implementaciones desarrolladas se han realizado de la manera más genérica posible para respetar el espíritu original de la arquitectura CERA-CRANIUM. El sistema de toma de decisiones y más en concreto el módulo encargado de calcular el Estado (tanto interno como externo) del bot y sus Motivaciones pueden modificarse de manera totalmente independiente no

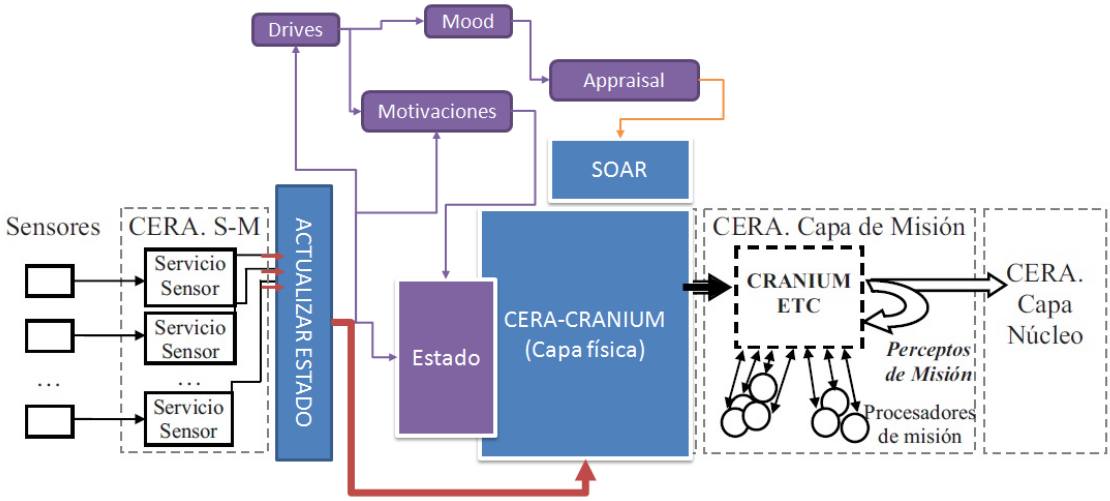

Figura 4.8: Integración del STD en el flujo bottom-up

sólo a la arquitectura en general sino también al módulo de comunicación con Soar, posibilitando si se desea en un futuro añadir o cambiar la información utilizada o su funcionamiento.

- **Aprendizaje por refuerzo:** Los valores de la tasa de aprendizaje ($\alpha = 0{,}3$) y del factor de descuento ($\gamma = 0{,}8$) de Sarsa son los valores por defecto de Sarsa.

- **Sistema de toma de decisiones:**

  1. Aunque en un principio se pensó en realizar el control del estado como un procesador CERA-CRANIUM, el hecho de que los procesadores compitan entre ellos complicaba su funcionamiento, ya que era posible que otro procesador "cogiera" un percepto (por ejemplo de presencia de enemigos) y el procesador de estado no pudiera procesarlo en el momento necesario. Esto provocaba un peligroso desfase entre el estado que el sistema enviaba al sistema de toma de decisiones y el verdadero estado del bot.
  2. Los valores de los parámetros estímulo motivacional ($w_i = 1$), valor del límite de activación de la motivación ($L_d = 2$) y *mood* ideal ($Wb_{ideal} = 100$) han sido fijados basándose en los resultados experimentales obtenidos en [39]. Se consigue de esta manera que el bot no se emocione con las novedades, pero tampoco las ignore.

- **Modelo de Riesgo:**

  1. Los valores utilizados para los pesos en el cálculo de la fórmula son $\alpha_{max} = 0{,}15$, $\alpha_{min} = 0{,}15$, $\alpha_{next} = 0{,}35$ y $\alpha_{prev} = 0{,}35$. La razón por la que se han establecido mayores pesos a las puntuaciones "siguiente" y "anterior" que a las correspondientes a las puntuaciones "máxima" y "mínima" de la partida, es porque en un comportamiento humano el jugador intentará quedar siempre en la mejor posición posible aunque no pueda ganar, de manera que sentirá más interés por cuánto le queda para subir o bajar un puesto en la clasificación de la partida.
  2. Para evitar sobrecargar la ejecución del bot, el valor de Riesgo se actualizará únicamente cuando recibamos el evento de que ha muerto otro jugador o cuando sea él mismo el que ha muerto (por otra parte, son estos los únicos casos en los que se actualiza la clasificación en las partidas DeathMatch).

3. El valor Riesgo tendrá siempre un valor Rϵ[1,2]. Se establecido el radio de selección de acciones inicial a la mitad de la activación máxima para evitar problemas al aplicar el modificador.

En el Anexo E se describen brevemente las clases y métodos más importantes utilizados en la implementación de CCBotSoar (acciones, perceptos, procesadores, etc).

4.8. Evaluación ConScale de CCBotSoar

En el siguiente cuadro resumen se recogen las habilidades cognitivas (CS) de la escala ConScale que se cumplen gracias a estas nuevas implementaciones. Se añade además la definición de los perfiles de comportamiento (BP) específicos para FPS definidos (Cuadro 4.6)

CS	**BP**	**Implementación**
3,7	El bot almacena en su memoria la posición de los botiquines y de los kits de munición. El bot va directamente a las posiciones recordadas cuando necesita munición o recuperar su salud.	Memoria de Objetos
4,2	El bot muestra comportamientos dirigidos y sostenidos hacia sus enemigos, como perseguirlos, dispararlos continuadamente o huir de ellos.	Sistema de Toma de Decisiones (Aprendizaje por Refuerzo)
4,3	Las acciones del bot que no contribuyen a las metas perseguidas son descartadas.	Sistema de Toma de Decisiones (Aprendizaje por Refuerzo)
5,1-5,3	Algunos comportamientos del bot se interrumpen debido a ciertas circunstancias y luego se retoman. Adicionalmente, el bot estima hasta qué punto las metas se están alcanzando en base a las estrategias empleadas. Los comportamientos más efectivos se repiten con más frecuencia que los comportamientos que suelen conllevar resultados pobres.	Sistema de Toma de Decisiones (Motivaciones - Aprendizaje por Refuerzo)
5,6	El bot genera de forma autónoma representaciones internas de alto nivel de los sucesos que tienen lugar a su alrededor.	Mapa de Peligro / Modelo de Riesgo
6,1-6,2-6,3	El bot entra en un estado determinado dependiendo de la evaluación que realiza del propio rendimiento. El comportamiento global del bot se ve modulado por el estado emocional en el que se encuentra.	Sistema de Toma de Decisiones (Motivaciones)

Cuadro 4.6: Avance en ConScale

Con esta nueva configuración podemos calcular, gracias a la herramienta calculadora de ConScale (http://www.consscale.com/es/calculadora/calculadora-fps.html) los parámetros del perfil cognitivo de CCBotSoar (Figura 4.9):

Aunque la implementación **satisface la mayoría de los requisitos de los niveles 4 y 5** (todos excepto uno en cada nivel) y algunos de nivel 6 (la mitad, para ser exactos) se clasifica como **nivel 3**. ConsScale requiere el completo cumplimiento de todas las habilidades cognitivas de un

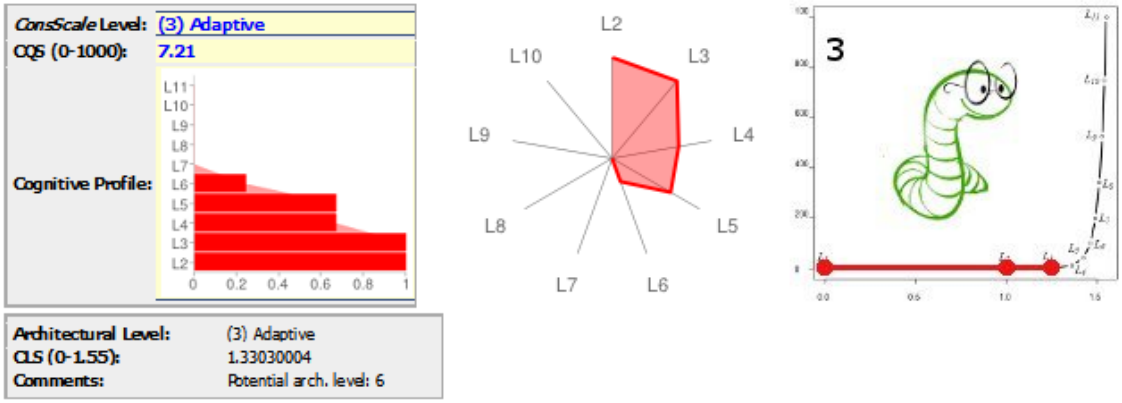

Figura 4.9: Perfil Cognitivo CCBotSoar

nivel i y también de todos los niveles inferiores para poder clasificar un agente como perteneciente al nivel i. El índice CQS de un agente puro de nivel 4 es 12.21. La puntuación de este agente (7.21) indica que existen características adicionales pero el agente está lejos de un agente de nivel 4 que puntuaría 12.21 o más.

Si bien puede parecer que este incremento no es significativo, debemos fijarnos en el valor CLS para comprender que, realmente, estamos más cerca de alcanzar el nivel 4 de lo que pensamos. El valor CLS (Cumulative Level Score) combina las puntuaciones de todos los niveles en una medida única que sigue una progresión logarítmica. El CLS de los niveles 3 y 4 es, respectivamente, 1.25 y 1.361. El valor CLS de nuestro bot alcanza el 1.33, lo que nos indica que nos encontramos a muy poca distancia del nuevo objetivo, que sería el nivel 4. La explicación es tan simple como que tanto en el nivel 4 como en el nivel 5, tan sólo necesitaríamos cumplir un único requisito añadido para completarlos.

Capítulo 5

Entrenamiento y experimentos

En este capítulo se explica el procedimiento utilizado para el entrenamiento del bot y los experimentos necesarios para comprobar si el avance en la escala ConScale se corresponde con un incremento en la "humanidad" del bot.

5.1. Entrenamiento

Como se ha mencionado en el Capítulo 4, nuestro bot implementa una memoria a largo plazo basada en el almacenamiento externo de las reglas de selección creadas por Soar durante el transcurso de una partida. Puesto que el diseño anterior del sistema no contemplaba la posibilidad del aprendizaje por refuerzo, nuestro bot se encuentra en un estado de *tábula rasa* con respecto a su sistema de toma de decisiones. Mediante este entrenamiento se pretende conseguir que nuestro bot tenga, a la hora de ser juzgado y comparado con otros, una memoria básica como jugador de Unreal Tournament 2004.

5.1.1. Entrenamiento específico

A lo largo del proceso de desarrollo y testeo de las nuevas implementaciones se hizo patente que debido a la integración del sistema basado en reglas el bot a veces actuaba de manera extraña, siendo su principal error visible el de no disparar a los enemigos aunque estuvieran a su alcance y simplemente apuntarles. La razón de este comportamiento es que nuestro sistema basado en reglas necesita un entrenamiento para aprender, al menos de una manera básica, como debe actuar en una serie de situaciones como podría ser la expuesta.

Precisamente es este comportamiento de pasividad ante la presencia de enemigos una de las características que podría causar una baja puntuación del bot en un concurso de "humanidad" como es el BotPrize.

Para solucionar este problema, el CCBotSoar ha recibido primero un entrenamiento específico en un mapa personalizado construido con el Unreal Editor (ver Anexo F). Se han realizado diez partidas en este escenario utilizando como enemigos diferentes tipos de bots "pasivos". Definimos un bot pasivo como un bot que no ataca a sus enemigos. En nuestro caso se han elegido dos bots ya predefinidos como arquetipos en el paquete de instalación de Pogamut para Unreal Tournament 2004: un bot vacío y estático que no hace nada (EmptyBot) y bot de navegación que recorre el mapa siguiendo los puntos definidos en este (NavigationBot). Al no ser atacado por el resto de contrincantes, nuestro bot tiene plena libertad para explorar todas las posibles acciones que desee

sin consecuencia y, en última instancia, reforzará los comportamientos de ataque a enemigos pues le reportarán un mayor beneficio que vagar por el escenario.

En el Cuadro 5.1 se explican los enemigos utilizados durante las partidas de entrenamiento específico:

Partida	Enemigos
1-3	3 EmptyBot
4	2 EmptyBot + 1 NavigationBot
5	1 EmptyBot + 2 NavigationBot
6-8	3 NavigationBot
9-10	3 NavigationBot + 1 EmptyBot

Cuadro 5.1: Enemigos durante partidas entrenamiento

El esquema de este mapa y una captura durante las sesiones de entrenamiento se encuentran en la Figura 5.1.

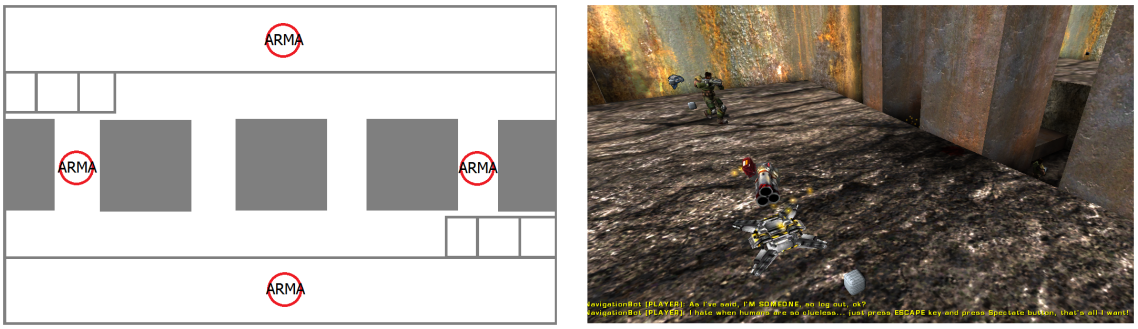

Figura 5.1: Escenario de Entrenamiento Especial (Esquema y Captura)

5.1.2. Entrenamiento general

Junto a este entrenamiento específico se ha realizado un entrenamiento más general haciendo competir al CCBotSoar en partidas contra bots genéricos de Unreal Tournament 2004. Un total de 20 partidas se han llevado a cabo durante esta fase de entrenamiento y en cada una de ellas, nuestro bot se enfrentaba a 4 bots controlados por el juego y a un jugador humano, el autor de este proyecto.

Se han elegido escenarios enfocados a la participación de como mínimo 4 jugadores y como máximo 8. Esto nos asegura un marco de experimentación lo suficientemente abierto como para que el bot pueda interaccionar no sólo con el resto de jugadores sino también con el entorno del mapa y lo suficientemente acotado como para que se mantenga la intensidad de la partida, lo que no sería posible si el escenario es tan grande que los encuentros entre jugadores son esporádicos.

Los enfrentamientos 1 contra 1 se han descartado como parte del entrenamiento pues nos interesa saber como se desenvuelve el bot en entornos dónde la información sensorial es fugaz y variable y que presenten retos a su estructura cognitiva, lo que no sucede en este tipo de escenarios.

5.2. Experimentos

Hemos comprobado teóricamente en la sección 4.8 que el nuevo bot CCBotSoar es, cognitivamente hablando, superior a CCBot2. Pero aunque la escala ConScale ha sido probada anteriormente, no deja de ser una escala teórica que ha demostrado en varias ocasiones no tener efectos inmediatos en la percepción del bot como "humana" desde una perspectiva exterior.

5.2.1. Preparación del sistema

Para la realización de este experimento se ha necesitado montar previamente la infraestructura necesaria. Esta infraestructura se podrá utilizar además posteriormente para el BotPrize universitario organizado por las universidades de Zaragoza, U-TAD (Madrid), Autónoma de Barcelona y Edith Cowan University (Perth, Western Australia).

La infraestructura establecida se compone de 4 equipos conectados a la red. Aunque en un principio se pensó en crear una red local para la conexión de los 4 equipos, el potencial uso global del sistema terminó por descartar esta opción.

Uno de los equipos servirá como servidor del sistema y los otros 3 serán serán los utilizados por los jueces para participar en las rondas del experimento/concurso. El servidor necesita, además de la instalación del juego Unreal Tournament 2004 que se realizará en todos los equipos del sistema, una base de datos que almacenará, mediante el uso de varios scripts, los resultados obtenidos durante las rondas. Para este fin se ha instalado XAMPP, ya que además del servicio MySQL para la creación de la base de datos, dispone de un servidor Apache desde el que se pueden ejecutar los scripts para generar los resultados.

La partida lanzada por el servidor se protegerá con una contraseña para evitar el acceso de usuarios ajenos al experimento/concurso. Para participar en la partida, los jueces simplemente tendrán que buscar el servidor de juego (llamado "BOTPRIZE") en la lista de servidores online que aparecen en la pestaña correspondiente de partidas multijugador de Unreal Tournament 2004 (*Join Game* $\rightarrow$ *Internet*) y buscar el tipo de juego *GameBotsDeathMatch*. Las partidas no tiene un mínimo de jugadores por lo que una vez introducida la contraseña de acceso entrará al juego.

Los bots que se quieran evaluar se pueden ejecutar tanto en la máquina servidor como conectándolos de manera remota.

Antes de proceder a la realización de los experimentos se realizaron pruebas para probar la robustez del sistema (estado de la red, conexiones de bots y jueces, recogida de resultados, etc) y se organizaron varios simulacros para enseñar a los jueces el funcionamiento del experimento.

5.2.2. Experimento y resultados

El objetivo del siguiente experimento es comprobar si realmente este avance en ConScale se corresponde con un incremento en la percepción de "humanidad" que tendrían otros jugadores humanos del CCBotSoar. Para ello se ha utilizado un planteamiento muy similar al utilizado en el BotPrize para determinar el ganador del concurso y comprobar si alguno de los participantes ha superado el test de Turing.

En líneas generales, el procedimiento para realizar el experimento es el siguiente:

- Se crea una partida utilizando las modificaciones de Unreal Tournament 2004 para poder emular las normas del concurso BotPrize (disponibles en la web). Estas modificaciones tienen como principal objetivo transformar el arma *LinkGun* para su utilización como herramienta de "marcado". Utilizando los disparos primario y secundario de este arma, se "marca" a los

rivales en la partida como “bot” o “humano” respectivamente. El juego almacena información sobre las “marcas” que los jugadores humanos aplican a los bots y a otros jugadores humanos.

- Se realizan 10 partidas de 5 minutos cada una en 10 escenarios distintos, y en ellas participaran 5 jugadores: 3 jugadores humanos, el CCBot2 y el CCBotSoar. Durante la partida ninguno de los jugadores humanos sabrá si sus enemigos son bots u otros jueces pues el propio servidor cambia los nombres de todos los participantes por otros genéricos para crear una interfaz ciega que permita realizar el experimento de manera correcta.
- Una vez terminada la ronda de partidas, se utiliza dos scripts PHP (también disponible en la web de BotPrize) que sintetizan los datos almacenados en el log especial.

En la Figura 5.2 se puede ver una captura realizada durante el experimento. Frente a los enemigos aparece la última “marca” asignada por el juez.

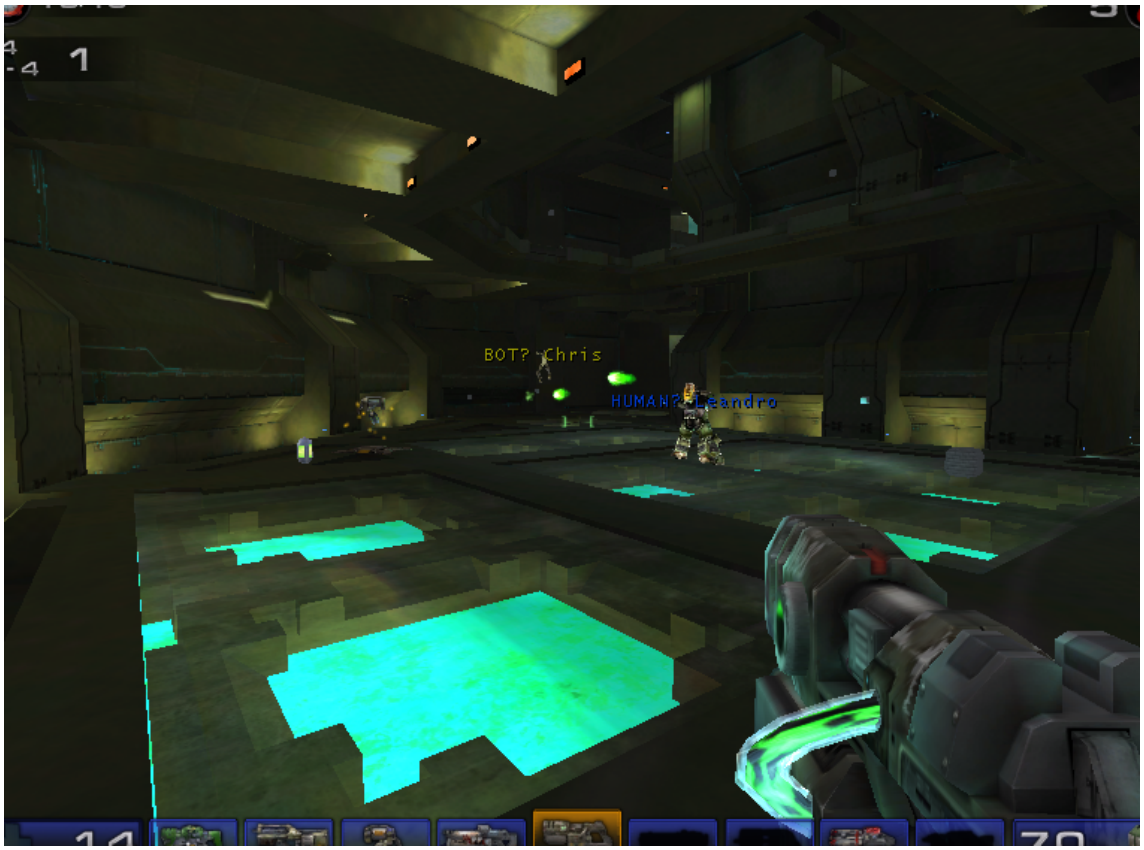

Figura 5.2: Captura del experimento

Los Cuadros 5.2, 5.3 y 5.4 presentan los resultados obtenidos tras la finalización del experimento. Los valores h-count y b-count reflejan el número de veces que un bot o un juez han sido marcados como “humano” o “bot”.

bot name	**h-count**	**b-count**	**humanness %**
CCBotSoar	15	44	25.4237 %
CCBot2	11	42	20.7547 %

Cuadro 5.2: Nivel de humanidad de los bots

player name	h-count	b-count	humanness %
juez1	50	36	58.1395 %
juez3	32	31	50.7937 %
juez2	15	20	42.8571 %

Cuadro 5.3: Nivel de humanidad de los jueces

human name	accuracy %
juez3	57.6923 %
juez2	56.0403 %
juez1	49.7110 %

Cuadro 5.4: Precisión de evaluación de los jueces

Como se puede apreciar, el porcentaje de humanidad del CCBotSoar es superior al del CCBot2 en casi un 5 %. Por lo tanto, queda comprobado experimentalmente que las implementaciones realizadas no sólo producen un aumento teórico en la humanidad del bot sino que también aumentan la humanidad real que otros jugadores humanos aprecian en él. Aún así, y como se esperaba, el nivel alcanzado no es suficiente para acercarse al porcentaje de humanidad que presentan los jueces, que rondan el 50 %.

El porcentaje de humanidad obtenido por el CCBot2 durante el BotPrize de 2010 (edición en la que resultó ganador) fue de un 31.81 %. La diferencia apreciable con el porcentaje obtenido durante este experimento se debe a que las condiciones del experimento, a pesar de ser similares, no son exactamente las mismas. En el concurso BotPrize 2010 participaron muchos más bots y por tanto muchos más jueces, de manera que las situaciones en las que es más identificable que el bot es efectivamente un bot, normalmente cuando un juez se encuentra a solas con el bot, son menos habituales que otras más caóticas en las que participan varios enemigos (bots y jueces) donde es mucho más complicado diferenciarlos.

Capítulo 6

Conclusiones

Este capítulo es una recapitulación de las conclusiones obtenidas durante la realización de este proyecto. Para ello se analizan los objetivos cumplidos y los resultados del proyecto, y se contemplará cuáles pueden ser las líneas de trabajo futuro. También se muestra una valoración sobre lo que ha supuesto el desarrollo del proyecto a nivel personal.

6.1. Resultados y objetivos cumplidos

Durante el transcurso del desarrollo del proyecto se han ido cumpliendo los objetivos marcados inicialmente en la propuesta:

- Se ha realizado la evaluación de la arquitectura CERA-CRANIUM según la definición de Vernon, Meta y Sandini[17]. Esta evaluación mostró que la arquitectura necesitaba algún mecanismo de adaptación.
- Se ha realizado también la evaluación del bot CCBot2 en la escala ConScale de habilidades cognitivas. Los requisitos propuestos para alcanzar los niveles 4, 5 y 6 sirvieron como hoja de ruta para las nuevas implementaciones.
- Se han implementado nuevas funcionalidades como la Memoria de Objetos, el Mapa Interno de Peligro, el Modelo de Riesgo y se ha desarrollado e integrado un Sistema de toma de Decisiones dentro de la arquitectura CERA-CRANIUM.
- Se ha realizado un análisis experimental del comportamiento del nuevo bot (CCBotSoar).
- Se han obtenenido conocimientos y experiencia en el uso de tecnologías y herramientas para el desarrollo software en la ingeniería, como son Java[1], SVN[2], así como aprender a trabajar con herramientas como son Pogamut, jSoar o los lenguajes que utilizan las diferentes arquitecturas cognitivas; y otras conocía pero no utilizaba habitualmente como los sistemas basados en reglas.
- Me he introducido en el ámbito de la investigación y experimentación de la inteligencia artificial.

[1] http://www.java.com
[2] http://es.wikipedia.org/wiki/Subversion

6.2. Trabajo futuro

La integración del Sistema de Toma de Decisiones dentro de la CERA-CRANIUM se ha realizado de manera que resulte totalmente transparente al contexto en el que se quiera utilizar, respetando así el concepto de la propia arquitectura.

Tras la realización de este proyecto se proponen a continuación algunas de las posibles líneas de trabajo futuras para mejorar el bot implementado (CCBotSoar) y la propia arquitectura cognitiva CERA-CRANIUM:

- Avances en la escala ConScale de habilidades cognitivas. A la vista de los posibles requisitos que se presentaron en 3.3.2 y que finalmente no fueron desarrollados, es muy posible que con la implementación correcta del requisito $CS_{4,4}$("*Capacidad básica de planificación: cálculo de las n siguientes acciones secuenciales*") pueda cumplirse también el requisito $CS_{5,5}$("*Capacidad avanzada de planificación teniendo en cuenta todas las metas activas*") por lo que el nuevo bot alcanzaría un nivel 5.
- Mejora y enriquecemiento de la información manejada en el Sistema de Toma de Decisiones para una representación más completa de los estados Interior y Exterior del agente así como de las Motivaciones utilizadas por el sistema como por ejemplo el Odio hacia ciertos jugadores que hayan matado a nuestro bot varias veces.
- Aplicación de la arquitectura CERA-CRANIUM a otros contextos para intentar averiguar su verdadero potencial cognitivo. Ya que su esctructura lo permite, podrían implementarse varios bots orientados a distintos objetivos y en distintos entornos (por ejemplo otros tipos de partida u otros juegos no FPS) y comparar sus resultados con otras arquitecturas que también puedan ser aplicadas a diversos marcos experimentales.

6.3. Valoración personal y problemas encontrados

El principio de este proyecto necesitó una ardua tarea de documentación sobre Pogamut y la arquitectura CERA-CRANIUM, la primera para familiarizarme con el funcionamiento y desarrollo de los bots y la segunda para conocer en profundidad el punto de partida del proyecto y sus posibles mejoras. Más tarde fue la implementación concreta de CCBot2 la que necesitó una revisión a fondo.

Una vez planteadas las potenciales implementaciones a realizar durante el proyecto el siguiente paso consistió en buscar la tecnología o tecnologías más idóneas para estos objetivos.

Este proyecto me ha permitido abordar el ámbito de la investigación y de la inteligencia artificial, ya que considero que la formación en Ingeniería Informática está más enfocada al desarrollo software y el mundo empresarial. Gracias a esto, dispongo de más conocimientos para saber hacia donde enfocar mi carrera profesional.

El hecho de que las cosas no siempre hayan funcionado como se esperaba, ha servido para aprender a enfocar los problemas desde una perspectiva más positiva. Casi todas las nuevas implementaciones desarrolladas han sufrido varios cambios radicales a lo largo del desarrollo porque las limitaciones del sistema o las propias ideas no desembocaban en los resultados deseados. No obstante, estos cambios ayudaron a un desarrollo mucho más unificado de todas las nuevas características implementadas.

Los problemas mas importantes encontrados a lo largo de la realización del proyecto han sido:

- El hecho de trabajar con una plataforma en desarrollo como es el caso de Pogamut, cuyos problemas de ejecución y constantes actualizaciones alargaron considerablemente la vida del proyecto.

- La falta de documentación por parte de las plataformas utilizadas tanto Pogamut como jSoar.
- Debido a la mala gestión de los recursos por parte de Pogamut, en ocasiones el cálculo de rutas de navegación no se hace correctamente, por lo que el comportamiento de dichos bots es erróneo e incluso se llegan a producir errores irrecuperables en su ejecución.

En cuanto a la realización del proyecto, cabe destacar la satisfacción de:

- Utilizar plataformas complejas y completamente desconocidas como eran Pogamut y la versión en java de Soar (jSoar).
- Haber contribuido a las investigaciones realizadas en el grupo GIGA de la universidad de Zaragoza para la utilización de arquitecturas cognitivas en el ámbito de los videojuegos y a la organización del BotPrize universitario 2013/2014.
- Haber adquirido conocimientos acerca de temas desconocidos de psicología e inteligencia artificial como son los utilizados para en la arquitectura cognitiva CERA-CRANIUM, la implementación CCBot2 y la escala de habilidades cognitivas ConsCale.

Glosario

- **Bot:** Agente autónomo virtual.
- **FPS:** Siglas del término inglés “First Person Shooter”. Genero de videojuegos de disparos en primera persona.
- **TCP/IP:** Modelo de descripción de protocolos de red. El modelo TCP/IP, describe un conjunto de guías generales de diseño e implementación de protocolos de red específicos para permitir que un ordenador pueda comunicarse en una red.
- **UnrealScript:** Lenguaje pensado exclusivamente para desarrollar contenido para juegos que usen el motor Unreal. Está basado en Java y C++. Es un lenguaje de programación orientado a objetos (OOP - Object Oriented Programming) lo que significa que está basado en los conceptos de clases y herencias. Es importante hacer notar que UnrealScript no tiene todas las características de un lenguaje de programación más completo.
- **APIs:** Interfaz de programación de aplicaciones (del inglés “Application Programming Interfaces”) es el conjunto de funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.
- **WME:** Elemento de la memoria de trabajo interna de Soar (del inglés “Working Memory Element”). En esta memoria de trabajo es donde se representa la situación actual del problema a resolver por Soar, y esta representación esta definida por los WME. Están compuestos por un identificador, un atributo y un valor (Ejemplo: “B1 es un bloque”. “B1 es rojo”).
- **ETC:** Espacio de Trabajo Compartido.
- **Input WME:** Son WME que vienen del input-link de Soar. A través del input-link se recibe la información del entorno externo con el que esta trabajando Soar. Los input WME definen esta información del entorno externo (Ejemplo: “Bot1 ataca Bot2”). Análogamente se pueden definir los **Output WME**.
- **Grounding:** Problema clásico de la filosofía del lenguaje. Se refiere al proceso por el que nuestras ideas o conceptos se relacionan con los hechos o cosas que existen (Ejemplo: la cadena de caracteres “gato” para el ordenador no significa nada, para nosotros es un animal con pelo y bigotes). A esta relación entre ”palabra” y ”objeto” al que se refiere, se denomina ”anclaje” o, “enraizamiento” (en inglés, “grounding”) del lenguaje.
- **Epistemología:** Teoría del conocimiento. Conjunto de métodos utilizados por el agente para obtener el conocimiento.
- **Filogenia:** desarrollo de un conjunto de agentes. En los sistemas cognitivos se llama filogenia a las características que tiene un agente por pertenecer a un tipo, es decir, que forman parte de su estructura básica.

- **Ontogenia:** desarrollo de un agente. En los sistemas cognitivos se llama ontogenia a las características que puede desarrollar un agente de manera complementaria a las que ya poseía por su filogenia.
- **Respawn:** regeneración automática del bot cada vez que lo matan en una partida y ésta aún no se ha acabado.

Bibliografía

[1] Langley, P., Laird, J.E., and Rogers, S. (2009). Cognitive architectures: Research issues and challenges. Cognitive Systems Research, 10:141-160. (document)

[2] Laird, J. E., Newell, A., and Rosenbloom, P. S. (1987). Soar: an architecture for general intelligence. Artificial Intelligence, 33(1):1–64. (document), 4.1.1, 4.1

[3] Gemrot, J., Kadlec, R., Bída, M., Burkert, O., Píbil, R., Havlícek, J., Zemcák, L., Simlovic, J., Vansa, R., Stolba, M., Plch, T., and Brom, C. (2009) Pogamut 3 Can Assist Developers in Building AI (Not Only) for Their Videogame Agents. Agents for Games and Simulations: Lecture Notes in Computer Science, 5920:1-15. 1

[4] Ray, D. (2009). jSoar: A pure Java implementation of Soar. In The 29th Soar Workshop. Ann Arbor, MI.

[5] Turing, A.M. (1950). Computing Machinery and Intelligence. Mind 49: 433-460.

[6] Arrabales, R., Ledezma, A., and Sanchis, A. (2009). Establishing a roadmap and metrics for conscious machines development. Proceedings of the 8th IEEE International Conference on Cognitive Informatics. 3.1

[7] Berridge, K. C. (2004). Motivation concepts in behavioural neuroscience. Physiology and Behaviour, 81:179–209. 4.3.3

[8] Sutton, R. S. and Barto, A. G. (1998). Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA, A Bradford Book. 4.3.1

[9] Rolls, E. (2003). Emotions in Humans and Artifacts, chapter Theory of emotion, its functions, and its adaptive value. MIT Press. 4.3.3

[10] R. Arrabales, A. Ledezma, A. Sanchis. (2009) Assessing and characterizing the cognitive power of machine consciousness implementations. (document), 1

[11] Grześ, M., and Kudenko, D. (2008). Robustness Analysis of SARSA(λ): Different Models of Reward and Initialisation. Proceeding AIMSA '08 Proceedings of the 13th international conference on Artificial Intelligence: Methodology, Systems, and Applications:144 - 156 4.3.1

[12] Arnold, M. (1960). Emotions and Personality. Nueva York. Columbia University Press. 4.3.2

[13] Schachhter, S. and Singer, J. (1962). Cognitive, social and physiological determinants of emotional state. Psychological Review, 59:379-399. 4.3.2

[14] F. J. Varela, “Whence perceptual meaning? A cartography of current ideas,” in Understanding Origins – Contemporary Views on the Origin of Life, Mind and Society, ser. Boston Studies in the Philosophy of Science, F. J. Varela and J.-P. Dupuy, Eds. Kluwer Academic Publishers, 1992, pp. 235–263. 2.1

[15] Arrabales Moreno, R. and Sanchis de Miguel, A. ”A Machine Consciousness Approach to Autonomous Mobile Robotics”. In: 5th International Cognitive Robotics Workshop. AAAI-06. Boston, MA. July 2006.

[16] Arrabales Moreno, R. Ledezma Espino, A. and Sanchis de Miguel, A. ”Modelling Consciousness for Autonomous Robot Exploration”. In 2nd International Work-Conference on the Interplay between Natural and Artificial Computation, IWINAC 2007. Lecture Notes in Computer Science Series, Vol. 4527. pp. 51-60.

[17] Vernon, D. Etisalat Univ. Coll., Sharjah Metta, G.; Sandini, G. “A Survey of Artificial Cognitive Systems: Implications for the Autonomous Development of Mental Capabilities in Computational Agents” in Evolutionary Computation, IEEE Transactions on, Volume 11, Issue 2 (document), 1.1, 1.4, 2.1, 2.1, 6.1

[18] Arrabales Moreno, Raúl. “Evaluation and Development of Consciousness in Artificial Cognitive Systems” Universidad Carlos III de Madrid. Departamento de Informática. Febrero 2011. (document), 1, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6

[19] Baars, B.J. 1997, ”In the Theatre of Consciousness: Global Workspace Theory, A Rigorous Scientific Theory of Consciousness”, Journal of Consciousness Studies, no. 4, pp. 292-309. 1, 3.1

[20] A. Clark, Mindware – An Introduction to the Philosophy of Cognitive Science. New York: Oxford University Press, 2001. 2.1

[21] Z. W. Pylyshyn, Computation and Cognition, 2nd ed. Bradford Books, MIT Press, 1984. 2.1

[22] E. Thelen and L. B. Smith, A Dynamic Systems Approach to the Development of Cognition and Action, ser. MIT Press / Bradford Books Series in Cognitive Psychology. Cambridge, Massachusetts: MIT Press, 1994. 2.1

[23] J. A. S. Kelso, Dynamic Patterns – The Self-Organization of Brain and Behaviour, 3rd ed. MIT Press, 1995. 2.1

[24] J. L. Krichmar and G. M. Edelman, “Principles underlying the construction of brain-based devices,” in Proceedings of AISB ’06 - Adaptation in Artificial and Biological Systems, ser. Symposium on Grand Challenge 5: Architecture of Brain and Mind, T. Kovacs and J. A. R. Marshall, Eds., vol. 2. Bristol: University of Bristol, 2006, pp. 37–42. 2.1

[25] H. Gardner, Multiple Intelligences: The Theory in Practice. New York: Basic Books, 1993. 2.1

[26] D. Vernon, “The space of cognitive vision,” in Cognitive Vision Systems: Sampling the Spectrum of Approaches, ser. LNCS (In Press), H. I. Christensen and H.-H. Nagel, Eds. Heidelberg: Springer-Verlag, 2006, pp. 7–26. 2.1

[27] W. J. Freeman and R. N´u˜nez, “Restoring to cognition the forgotten primacy of action, intention and emotion,” Journal of Consciousness Studies, vol. 6, no. 11-12, pp. ix–xix, 1999. 2.1

[28] C. von Hofsten, "An action perspective on motor development," Trends in Cognitive Science, vol. 8, pp. 266–272, 2004. 2.1

[29] —, "On the development of perception and action," in Handbook of Developmental Psychology, J. Valsiner and K. J. Connolly, Eds. London: Sage, 2003, pp. 114–140. 2.1

[30] G. Sch¨oner, "Development as change of dynamic systems: Stability, instability, and emergence," in Toward a New Grand Theory of Development? Connectionism and Dynamic Systems Theory Re-Considered, J. P. Spencer, M. S. C. Thomas, and J. L. McClelland, Eds. New York: Oxford University Press, 2006. 2.1

[31] W. D. Gray, R. M. Young, and S. S. Kirschenbaum, "Introduction to this special issue on cognitive architectures and human-computer interaction," Human-Computer Interaction, vol. 12, pp. 301–309, 1997. 2.1

[32] F. E. Ritter and R. M. Young, "Introduction to this special issue on using cognitive models to improve interface design," International Journal of Human-Computer Studies, vol. 55, pp. 1–14, 2001. 2.1

[33] A Survey of Cognitive and Agent Architectures, http://ai.eecs.umich.edu/cogarch0/. 2.1

[34] P. Langley, "An adaptive architecture for physical agents," in IEEE/WIC/ACM International Conference on Intelligent Agent Technology. Compiegne, France: IEEE Computer Society Press, 2005, pp. 18–25. 2.1

[35] M. Jones and D. Vernon, "Using neural networks to learn hand-eye co-ordination," Neural Computing and Applications, vol. 2, no. 1, pp. 2–12, 1994. 2.1

[36] J. Weng, "Developmental robotics: Theory and experiments," International Journal of Humanoid Robotics, vol. 1, no. 2, pp. 199–236, 2004. 2.2

[37] M. P. Shanahan, "A cognitive architecture that combines internal simulation with a global workspace," Consciousness and Cognition, 2006, to Appear. 3.3.1

[38] M. P. Shanahan and B. Baars, "Applying global workspace theory to the frame problem," Cognition, vol. 98, no. 2, pp. 157–176, 2005. 3.3.1

[39] Gavalda Pina, Angel "Control de un agente inteligente basado en una arquitectura cognitiva para el entorno del videojuego Unreal Tournament 2004" Proyecto Fin de Carrera. Especialidad en Ingenieria en Informatica. Escuela de Ingeniería y Arquitectura, Universidad de Zaragoza, Julio, 2012. 4.3, 2

Parte II

Anexos

Anexo A

Manual de Pogamut 3

A.1. Instalación y Servidor

A.1.1. Instalación

En esta sección se describe el proceso de instalación, así como el software necesario para la misma.

Prerrequisitos

- Una copia del videojuego Unreal Tournament 2004 (UT2004)
- Unreal Engine 2 Runtime (UE2) and Unreal Development Kit (UDK)
  - UE2 es gratuito para uso no comercial. Es posible descargar la versión Demo de http://apacudn.epicgames.com/Two/UnrealEngine2Runtime22262002.html
  - UDK is una versión libre de Unreal Engine 3. Está incluido en el paquete de instalación de Pogamut 3.
- JDK y Netbeans
  - Descarga conjunta en http://www.oracle.com/technetwork/java/javase/downloads/jdk netbeans- jsp-142931.html
- Pogamut 3
  - Descargar "Pogamut 3 Java instaler full" en la sección Downloads de la página oficial http://diana.ms.mff.cuni.cz/main/tiki-index.php?page=Download

Proceso de Instalación

Para no tener problemas en el proceso de instalación de todos los elementos que necesitamos, seguir el siguiente orden, ya que al instalar Pogamut se nos pedirá la ubicación de UT2004, UE2 y Netbeans.

1. Instalar el videojuego Unreal Tournament 2004.

2. Instalar JDK y NetBeans.

    - En Windows 7 instalar Netbeans directamente en C:/ en lugar de en la ubicación por defecto, debido a problemas con los permisos.

3. Instalar Unreal Engine 2 Runtime.

4. Por último, instalar Pogamut.

    - Realizar la instalación como aparece por defecto, es decir, con todos los elementos seleccionados, ya que todos son necesarios para que todo funcione correctamente.
    - Si hemos instalado los programas en la ruta por defecto, el instalador de Pogamut los detectará automáticamente. En caso contrario tendremos que buscar la ruta en la que hayamos instalado el programa requerido por el proceso de instalación.

A.1.2. Ejecución del bot en UT2004

En esta sección se describe cómo configurar correctamente el servidor de UT2004, de forma que luego podamos conectar nuestro bot al mismo sin problemas. También se describe el proceso de conexión de dicho bot al servidor utilizando NetBeans.

Configuración del servidor

Para crear un servidor en UT2004, la forma más sencilla es hacerlo desde el propio juego. Para ello, abrimos UT2004 y seleccionamos la opción Alojar Partida. Una vez hecho esto, debemos seleccionar el tipo de juego. Para que éste sea compatible con nuestro bot, debemos elegir uno de los tipos de juego personalizados, situados al final de la lista (Figura A.1). Lo más común es seleccionar el tipo "GameBots DeathMatch".

Una vez hecho esto, configuramos el juego a nuestro gusto. En la sección "Reglas de servidor" debemos tener siempre en cuenta las siguiente consideraciones (Figura A.2):

- Activar la opción "Servidor LAN" para mejorar el rendimiento si estamos trabajando con ordenadores conectados en LAN.
- Deseleccionar la opción "Anunciar servidor" para evitar problemas con copias ilegales de UT2004.
- Activar "Ignore UTAN Bans" para evitar problemas con copias ilegales de UT2004.

Por último, podemos crear dos tipos de servidores:

- Mixto: nada más crear el servidor, pasaríamos directamente a jugar en él y, posteriormente, podríamos conectar otros bots contra los que jugaríamos. Esta opción no permite visualizar los acontecimientos que ocurren en el servidor, como es el caso del "servidor dedicado"
- Dedicado: este tipo de servidor permite visualizar en modo texto la configuración del servidor y los acontecimientos que ocurren, tales como la conexión de un jugador o bot, la muerte y resurrección de los mismos, etc. Si queremos, desde el mismo ordenador, conectarnos para jugar en el servidor, sólo tenemos que ejecutar otra vez UT2004 y conectarnos a él como jugadores.

Figura A.1: Modos de juego GameBots

Conexión del bot al servidor

Lo primero que debemos hacer es abrir nuestro proyecto en NetBeans. Una opción inteligente sería construir nuestro bot sobre "EmptyBot", un proyecto de ejemplo incluido por Pogamut, que sería el equivalente al "Hello world" de los bots. Para ello seleccionamos "nuevo proyecto" en NetBeans, seleccionamos 00-Emptybot situado en la carpeta Samples/Pogamut UT2004 y le damos un nombre. Nuestro proyecto quedará guardado por defecto en Documentos/NetBeansProyects. La figura A.3 muestra como acceder al código fuente de dicho proyecto.

A continuación, en la pestaña "Services" hacemos click derecho sobre "UT2004 Servers" y seleccionamos "Add server" (Figura A.4). En el diálogo que aparece (Figura A.5) elegimos un nombre para el servidor y, en cuanto a la URI, tenemos dos opciones:

- Si el servidor está en el mismo ordenador desde el cual vamos a conectar el bot, escribimos localhost (si localhost no funciona probar con 127.0.0.1:3001 en su lugar).
- Escribir la IP del ordenador donde se encuentra el servidor.

Una vez hecho esto, hacemos click en Close y todo estará preparado para ejecutar el bot sobre el servidor. En caso de aparecer un triángulo amarillo con una exclamación sobre el servidor que hemos creado, revisar la configuración del mismo, ya que posiblemente no tengamos problemas para acceder a él desde el ordenador donde está alojado el servidor pero sí desde uno externo.

Por defecto, siempre aparecerá el puerto 3001 al poner la IP. Ésto se debe a que los puertos por defecto son: 3000 para BotConnection, 3001 para ControlServer y 3002 para SpectatorConnection

Ahora, el servidor esta en ejecución y la IDE sabe como conectarse a él. Para ejecutar el bot simplemente corremos el bot mediante el botón con el triángulo verde de "play". Si todo funciona, el bot se conectará al servidor y Netbeans nos mostrará la información referente a su ejecución. Si queremos inspeccionar el bot desde el juego, tenemos dos opciones.

Figura A.2: Configuración del servidor

- Abrir el juego y conectarnos al servidor como jugadore o espectador.
- Desde NetBeans, hacer click derecho sobre el servidor y seleccionar Spectate (Figura A.6).

Por último, GAMEBOTS2004 incluye funcionalidades especiales, tales como la visualización de los navegadores y las líneas de movimiento, los rayos para detectar obstáculos, etc., los cuales podemos activar y desactivar a nuestro antojo mediante un menú al que podemos acceder presionando ALT + H durante la ejecución del juego (Figura A.7).

A.2. Modos de movimiento del bot

Existen dos maneras prefijadas para tratar el movimiento de los bots.

1. **Navegación:** El bot se mueve eligiendo un punto de navegación en el mapa y calculando la ruta a seguir para alcanzar dicho punto de navegación (navpoint).
2. **Raycasting:** El bot se mueve basándose en la geometría del mundo, analizándola a través de rayos en busca de intersecciones.

A continuación se explica en detalle ambas implementaciones, las cuales podrían ser usadas en paralelo, creando un bot mas completo.

A.2.1. Bot de Navegación

El mapa está cubierto por nodos llamados puntos de navegación (navpoints). En teoría cada navpoint está situado en un sitio seguro y alcanzable por el bot. Los puntos conectados están

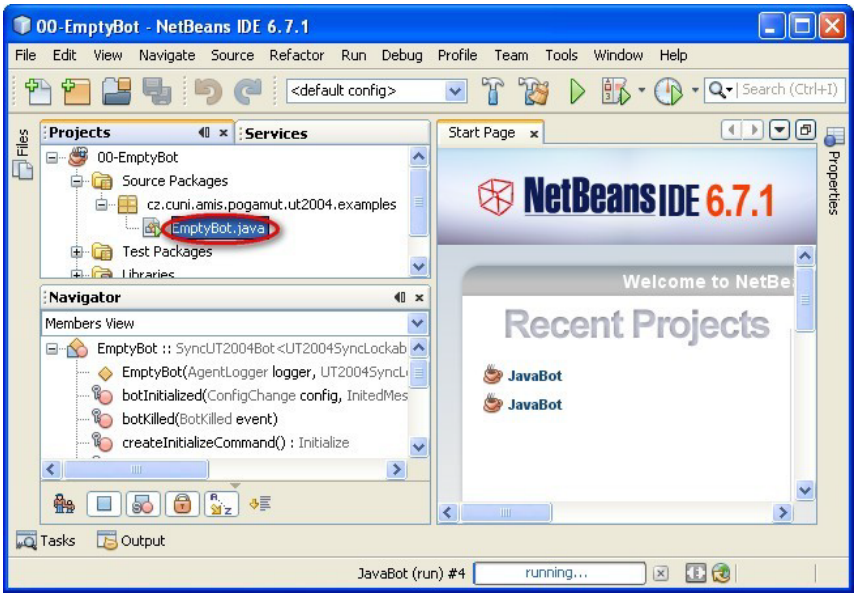

Figura A.3: Código en NetBeans

unidos por una línea. Para que el bot sea capaz de ir de un sitio A a un sitio B, vamos a necesitar una planificador de ruta: Path planner.

En Pogamut existen distintos interfaces de Path planner, siendo UT2004A StarPathPlanner el planificador por defecto. Este planificador usa el algoritmo A* para el cálculo de rutas, siendo exactamente el mismo que el de los bots nativos de UT. También existe otra implementación declarada en FloydWarshallPathPlanner, la cual precalcula todas las rutas posibles entre todos los nodos al principio, lo cual tiene un coste inicial considerable. Cabe decir que podemos implementar nuestro propio algoritmo planificador.

Una vez tenemos la ruta calculada ya solo nos queda ejecutarla, de ello se encarga el Path executor. Este módulo contiene el Path navigator que es el que se encargar de recorrer la ruta calculada anteriormente, evitar obstáculos, abrir puertas, esperar ascensores,etc. El navegador por defecto es UT2004PathNavigator. En este caso, también podemos crear nuestro propio navegador.

Veamos un ejemplo de implementación:

Algoritmo A.1 Ejemplo de bot de navegación

```
protected void goToRandomNavPoint() {
targetNavPoint = pickRandomNavPoint();
// find path to the random navpoint, path is computed asynchronously
// so the handle will hold the result onlt after some time
IPathFuture<ILocated> pathHandle = pathPlanner.computePath(info.getLocation(), targetNav-
Point);
// make the path executor follow the path, executor listens for the
// asynchronous result of path planning
pathExecutor.followPath(pathHandle);
}
```

En el ejemplo podemos ver como crea un punto de navegación al azar, y encontramos la ruta hasta dicho punto. Una vez la tenemos con followPath hacemos que el bot siga la ruta calculada.

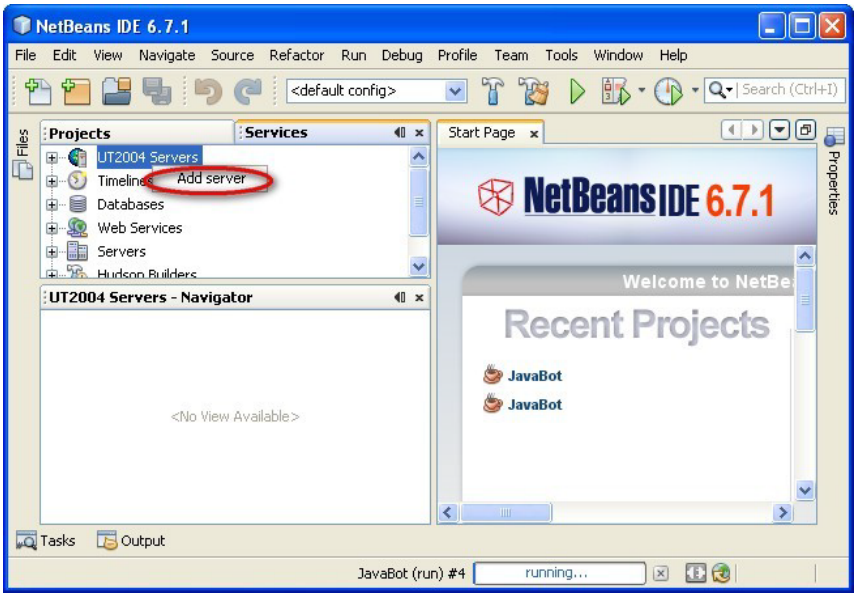

Figura A.4: Añadir servidor UT2004 a NetBeans

Podemos declarar un listener dentro del Path Executor, para ser notificados acerca del estado del bot mientras ejecuta la ruta. Esto será útil para saber si el bot se ha atascado, si el bot ha llegado a su destino o incluso si la ejecución de la ruta ha fallado.

Para hacer un uso más simple y potente de la navegación a través de navpoints del bot, podemos usar la libreria Jungigation. Las principales ventajas de son:

- Exclusión de líneas prohibidas: Así evitaremos el cálculo de una mala ruta.
- Planificación avanzada basada en los objetos del mapa
- Saber exactamente el tiempo en segundos de la ruta.

El principal añadido de esta librería es que el planificador de ruta solo usa caminos que han sido probados y que se sabe con seguridad que el bot puede ir por ellos. Para ello hemos tenido que previamente usar una utilidad para calcular dichas rutas, por lo que esta funcionalidad no será interesante a no ser que implementemos un bot con aprendizaje.

A la hora de testear nuestro bot con navegación, podemos seguir a través de UT a nuestro bot, haciendo visibles en el mapa (ALT+G) los puntos y líneas de navegación. Otra forma es visualizar el mapa de navegación desde Netbeans.

A.2.2. Bot con raycasting

Como hemos dicho anteriormente el raycasting se base en rayos, a través de los cuales veremos sus intersecciones con el mundo/mapa. Para usar raycasting en nuestro bot, debemos seguir tres pasos:

1. **Activar raycasting y visualización de rayos:** podemos hacerlo de dos formas, o bien computar continuamente las intersecciones de los rayos con el mundo (Autotrace=true) o bien definir nuestros rayos y calcular solo las intersecciones que nos interesen (DrawTraceLine=true).

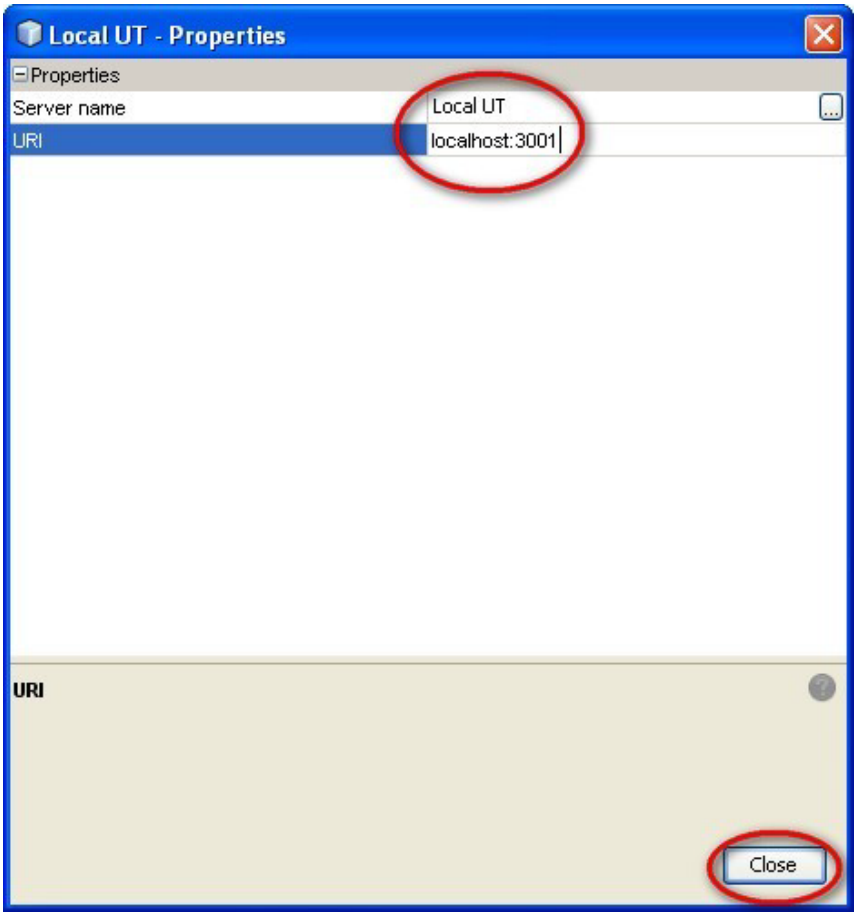

Figura A.5: Host del servidor UT2004 en NetBeans

2. **Inicializar rayos:** definimos los rayos que deseemos a través de AddRay, pasándole como parámetros principales el nombre, el vector de dirección y la longitud del rayo. Podemos crear tantos rayos como queramos.

3. **Manejar resultados:** una vez que el rayo ha sido lanzado, UT nos devuelve el resultado en un mensaje. Podemos tratarlos de dos maneras: Creando un listener y que nos notifique cuando un rayo ha recibido algo del entorno o bien comprobando la presencia de algún objeto periódicamente, por ejemplo en el método logic().

Cuando ya hayamos sido notificados, definiremos las funciones de actuación que nos interesen y el objeto de AutoTrace ray será actualizado automáticamente.

A.3. Implementación del bot

Crear un bot Una vez que ya tenemos el servidor conectado a Netbeans es el momento de empezar a programar el bot. Para eso lo primero es entender que nos da Netbeans con su plantilla básica, y a partir de ahí desarrollar nuestro propio código.

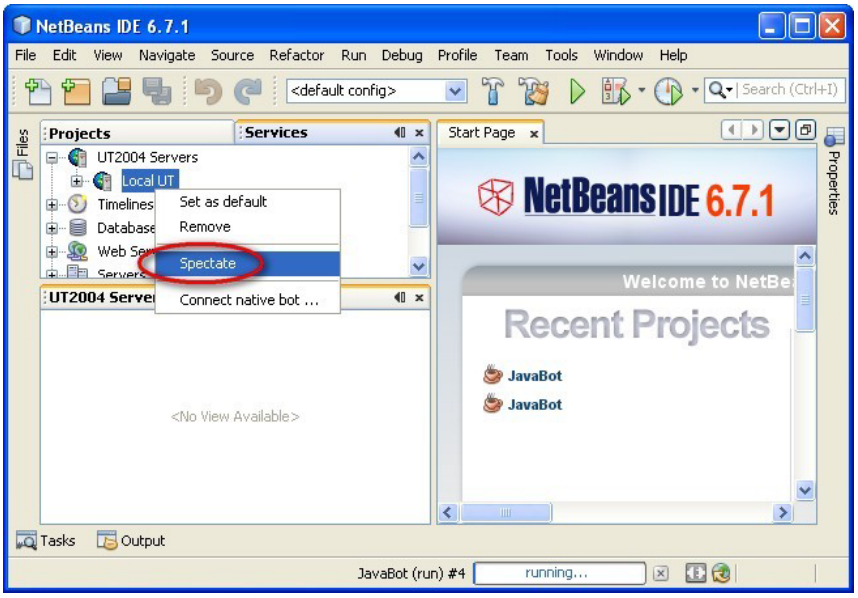

Figura A.6: Modo espectador en UT2004 desde NetBeans

A.3.1. Clases principales

En este apartado vamos a explicar las clases esenciales para crear un bot, desde la que se encarga de dar vida al bot a la que le permite moverse.

cz.cuni.amis.pogamut.ut2004.utils.SingleUT2004BotRunner

Esta clase es la encargada de lanzar un thread de ejecución para hacer correr al bot. Se sitúa dentro del main principal. Primero se establecen los parámetros con el servidor (host y port). Pueden ser de manera manual o que sea la propia clase que los encuentre. Después de establecer los parámetros se llama a startAgent() que es la encargada de lanzar el thread de ejecución. Se acaba cuando se elimina al bot de la partida.

cz.cuni.amis.utils.exception.PogamutException

En todas las funciones donde se ejecutan instrucciones de pogamut se añade esta clase para capturar excepciones. Únicamente se encarga de tratar cualquier excepción de pogamut que se pueda producir dentro de la función. En las funciones iniciales solo está definido en main() y en logic()

cz.cuni.amis.pogamut.ut2004.bot.impl.UT2004BotModuleController

Esta clase contiene el controlador más avanzado del sistema. Tiene todos los módulos útiles para manejar el bot. Es la clase principal. Antes de pasar a explicar los módulos vamos a comentar las funciones iniciales del bot que se encargan de establecer la secuencia inicial de ejecución. Esta secuencia inicial de ejecución se corresponde a este protocolo de GameBots2004:

1. Llama a *this.prepareBot()*

Figura A.7: Consola del comandos con opciones propias de Pogamut 3 para UT2004

2. → Recibe un mensaje HELLO_BOT, siendo el primer mensaje dentro de UT2004, el cual se encarga de pedir la conexión de un bot con el servidor. Si el servidor está lleno se termina la conexión.

3. ← Envía un mensaje READY. Como respuesta el servidor te envía al juego un mensaje NFO con información sobre la partida.

4. Ahora es cuando se establece la comunicación con GameBots2004.

5. → Captura el evento InitCommandRequested confirmando que la inicialización esta preparada.

6. Ejecuta this.getInitializeCommand().

7. ← Envía el comando INIT que se ha creado en la función anterior, con los parámetros iniciales del bot.

8. → Recibe el mensaje ConfigChange.

9. →Recibe el mensaje InitedMessage.

10. Llama a *this.botInitialized()*.

11. . . . se recibe el primer mensaje SLF mostrando la información que el bot ya puede conocer de sí mismo, al haber sido introducido ya en el sistema. También recibe el mensaje END para terminar la comunicación síncrona con el servidor.

12. Ejecuta *this.botSpawned()*.

13. El bot pasa a correr dentro del entorno por medio de la función *logic()*.

Todos estos pasos se explican más en detalle ahora al conocer las funciones iniciales que se ejecutan en este protocolo de comunicación.

- **prepareBot():** Se ejecuta antes de conectar al bot con el entorno y después de construir el UT2004Bot, es el sitio idóneo para inicializar los módulos de pogamut que se vayan a usar mas adelante.
- **getInitializeCommand():** Se utiliza para darle al agente las propiedades iniciales tales como nombre, localización inicial, *skin*, etc. Este método es usado por pogamut para obtener el objeto Initialize que contiene todos los parámetros generales del bot.
- **botInitialized(GameInfo info, ConfigChange config, InitedMessage init):** Primera vez que el bot tiene información sobre el mundo, aunque todavía no ha sido creado. Se llama a esta función una vez que el servidor ha enviado el mensaje INITED (clase de java: InitedMessage). Esto significa que el comando INIT ha sido ejecutado correctamente y la comunicación entre el bot y el servidor es correcta. En este momento ya es posible establecer nuevas comunicaciones entre el bot y el servidor, aunque el bot no haya sido todavía creado en el entorno, por lo que los comandos de movimiento no se pueden ejecutar, pero sí los de información del juego (GameInfo), configuración del sistema actual (ConfigChange) y mensajes de inicialización (InitedMessage).
- **botSpawned(GameInfo info, ConfigChange config, InitedMessage init, Self self):** Se ejecuta cuando aparece el bot en el juego, justo al entrar en el entorno del juego. Significa que la representación gráfica del bot ya es visible en el entorno.
- **logic():** Método principal del sistema, se ejecuta periódicamente por un thread interno asociado al agente. Esto le da la posibilidad al agente de ser proactivo y le permite actuar sin necesidad de estímulos externos. Todo lo que se ejecute dentro del método debe ser procesado rápidamente ya que se ejecuta cada 0.25 segundos.

A.3.2. Clase ModuleController

En esta sección vamos a centrarnos en todos los componentes contenidos en la clase *UT2004ModuleController*.

Doce de sus componentes son módulos completos y complejos que permiten recoger y enviar información sobre el bot y sobre su entorno. Estos son dichos módulos:

- **Players:** Información sobre el resto de jugadores. Todos los objetos de este tipo se autoactualizan a lo largo de la partida hasta que son destruidos. Este módulo permite al agente conocer toda la información sobre el resto de jugadores, informar sobre la posición, la visibilidad, el armamento, salud, bonificadores, etc.
- **Game:** Información general sobre el juego. Con este módulo se puede saber: el tipo de partida (DeathMatch, equipos, captura de bandera, etc), el nombre del mapa, tiempo total de la partida, número máximo de equipos y salud, armadura y adrenalina iniciales, etc.
- **Info:** Información sobre el paradero del agente. Este módulo permite saber todo sobre el agente: nombre, ID, equipo, localización, velocidad actual, salud, armadura, adrenalina, etc.
- **Items:** Información sobre los objetos que se encuentran en el mapa. Se pueden obtener: mapas con todos los *items* del escenario y diferentes filtros (por ejemplo, sólo los visibles, sólo de un tipo (salud, armas), etc).
- **Senses:** Información sobre lo que siente y percibe el agente. Permite controlar todo lo que afecta al bot (por ejemplo, informar dónde fue golpeado, el último elemento recogido, el último daño causado, último daño recibido, si ve proyectiles que vienen hacia él, etc).

- **Weaponry:** Información sobre el inventario del agente. Este módulo permite al bot saber la cantidad de munición del arma actual (disparo primario y secundario) o de todas las armas del inventario, características del arma que está usando, etc.
- **Shoot:** Clase que contiene las acciones de disparo avanzadas que puede realizar el agente (por ejemplo, disparar arma primaria o secundaria a un objetivo, o a una localización, recargar arma, dejar de disparar, etc).
- **Move:** Clase que contiene opciones de movimiento para el agente y que puede utilizar conjuntamente con el resto de opciones de movimiento (*Raycasting*, *navigation points*, etc.) Dispone de métodos para esquivar disparos u objetivos, saltar, ir a una localización determinada, correr, detenerse, poner el foco de atención en un objeto o jugador, etc.
- **PathPlanner:** Junto con *pathExecutor* es otra forma de moverse. Se encarga de calcular la ruta de acceso a un punto usando como apoyo los puntos de navegación del mapa del juego. A partir de dos puntos del mapa es el responsable de encontrar la ruta que se debe seguir para ir de un punto a otro.
- **PathExecutor:** Se encarga de ejecutar el *pathPlanner* que ha sido calculado previamente siguiendo la ruta.
- **ListenerRegistrator:** Es el método encargado de autoinicializar los listeners por medio de *AnnotationListenerRegistrator*. Se encarga de proporcionar una manera sencilla y práctica de registrar los listeners.
- **Descriptors:** Es un módulo sensorial que informa de las características generales de cualquier Item, en este caso son la categoría a la que pertenece (Weapon, Ammo, Health, Armor, Shield, Adrenaline, Other) y el grupo al que pertenece (Assault_Rifle, Minigun, Health, Mini_Health, Small_Armor, Super_Armor, Adrenaline, Udamage, Key, etc).
- **Config:** módulo de memoria especializado en la configuración del agente dentro de UT2004 (velocidad de rotación, retraso de envíos síncronos, raytracing, invulnerabilidad, respawn automático, recogida automática de ítems, etc).
- **Raycasting:** soporte para la creación de rayos que permiten detectar objetos en la escena. El rayo se pone verde cuando el camino está libre y rojo cuando hay obstáculos en la dirección y longitud del rayo. Lo que consiguen los rayos es que el bot circule solo por las zonas donde los rayos son verdes.
- **Body:** maneja los comandos que pueden ser usados como entrada de datos en el bot.
    - **Action:** Proporciona los comandos de acción del bot. o Arrogar armas o Iteración con elementos del entorno (recoger ítems) o Otros comandos que no han podido ser clasificados en otras categorías (renacer).
    - **Communication:** Proporciona los comandos de comunicación. o Enviar mensajes (privados, globales o de equipo). o Recibir mensajes (privados, globales o de equipo).
    - **ConfigureCommands:** Permite cambiar los atributos del bot. Nombre, skin, velocidad de movimiento y de rotación, invulnerabilidad, etc.
    - **AdvancedLocomotion:** Módulo "move", explicado a continuación.
    - **AdvancedShooting:** Módulo "shoot", explicado a continuación.
    - **SimpleRayCasting:** Control básico de rayos. Similar al módulo comentado anteriormente de "rayCasting". Su funcionalidad es la misma, el uso de rayos para detectar por donde puede moverse el bot y por donde no puede hacerlo.

- **Random:** módulo generador de números aleatorios especialmente útiles a la hora de llevar a cabo una toma de decisiones.

- **Act:** forma parte de IAct, que es el entorno encargado de gestionar todas las acciones y comandos que se pueden realizar en este mundo. Ligado a *World* (IVisionWorldView) ya que los dos juntos forman todo el entorno del juego, uno se encarga de las acciones (IAct) y el otro del entorno y los eventos que suceden en él (IVisionWorldView).

- **World:** forma parte de IVisionWorldView, que es el entorno encargado de gestionar el entorno y todos los eventos que se producen en él. Ligado a *Act*.

A.3.3. Otros comandos interesantes

Anotación de java: @JProp

Anotación que se coloca delante de las variables java que quieras monitorizar en tiempo de ejecución. En tiempo de ejecución podemos acceder a estas variables, para modificarlas y ver su comportamiento en el juego, desde "Servidores → UT2004 servers → Name Server → Bot pogamut → Introspection → Properties.

GBCOMMANDS

Esta librería *cz.cuni.amis.pogamut.ut2004.communication.messages.gbCommands* contiene todos comandos que se pueden introducir en GameBots. Al usar java con Netbeans no es necesario introducir los comandos de manera literal. Java nos permite tener una clase para cada uno de los comandos que se pueden manejar. Con esto cada vez que se use en una clase un comando

acciones que se pueden realizar en GameBots pero java nos da una interfaz más compleja con la que poder utilizarlos sin tener que preocuparnos de tener que llamar a cada uno de los comandos de manera literal.

Ahora vamos a poner la lista de todas las clases de comandos, la mayoría se entienden por si mismas así que no es necesario explicarlas, aun así dentro del javadoc de pogamut hay un extensa explicación de los campos y las funciones que componen cada una de las clases (una por comando). Además en el javadoc se puede ver con que comando de GameBots está relacionada cada clase.

- Act
- AddBot
- AddInventory
- AddRay
- Combo
- CommandPlayer
- Configuration
- ConfigurationObserver
- Console
- ContinuousMove
- DialogBegin
- DialogCancel
- DialogEnd
- DialogItem
- DisconnectObserver
- Dodge
- DriveTo
- EndPlayers
- EnterVehicle
- FactoryUse
- FastTrace
- GetAllInvetories
- GetAllNavPoints
- GetAllStatus
- GetGameInfo
- GetItemCategory
- GetMaps
- GetPath
- GetPlayers
- GetSelf
- GetSpecialObjects
- GetVisibleObjects
- GiveInventory
- ChangeAttribute
- ChangeMap
- ChangeTeam
- ChangeWeapon
- CheckReachability
- Initialize
- InitializeObserver
- Jump
- Kick
- LeaveVehicle
- MovePassword
- Reply Pause
- Pick Ping
- PlaySound
- Quit
- Ready
- Record
- RemoveRay
- Respawn
- Rotate
- SendMessage
- SetCrouch
- SetDialog
- SetGameSpeed
- SetLock
- SetPassword
- SetPlayerControl
- SetRoute
- SetSendKeys
- SetSkin
- SetWalk
- Shoot
- ShowText
- SpawnActor
  StartAnimation
- StartPlayers
- Stop
- StopRecord
- StopShooting
- Throw
- Trace
- TurnTo

A.4. Eventos

A.4.1. Interacción con el mundo

Las interfaces *IworldView* junto con *Iact* representan la API básica para acceder al mundo.

Información que recibimos (Listeners)

La interfaz *IWorldView* (cz.cuni.amis.pogamut.base.communication.worldview) nos ofrece tanto los sentidos del bot y como memoria simple. El mundo es representado por:

- Objetos (IworldObject): cz.cuni.amis.pogamut.base.communication.worldview.object ∘ AliveMessage ∘ AutoTraceRay ∘ BombInfo ∘ ConfigChange ∘ DominationPoint ∘ FlagInfo ∘ GameInfo ∘ IncomingProjectile ∘ InitedMessage ∘ Item ∘ ItemCategory ∘ Mover ∘ MyInventory ∘ NavPoint ∘ Player ∘ Self ∘ TeamScore ∘ Vehicle.
- Eventos (IWorldEvent): cz.cuni.amis.pogamut.base. communication.worldview.event:
  - Eventos de Objetos (IWorldObjectEvent): cz.cuni.amis.pogamut.base. communication.worldview.object
  - Eventos no asociados a ningún objeto (utilizados directamente de IWorldEvent). La lista de eventos es la siguiente. Observar que esta lista cuenta también con los eventos pertenecientes a objetos, ya que podríamos querer, por ejemplo, ser conscientes de todos los objetos que apareciesen en pantalla, para lo que usaríamos WorldObjectAppearedEvent.

En el siguiente apartado se muestra una información más detallada de los eventos pertenecientes a IWorldEvent.

Cuando queramos utilizar uno de estos eventos u objetos debemos incluir su correspondiente librería. Para ello consultar en la API la librería concreta de cada uno.

Una vez explicado qué información podemos obtener del mundo y de qué manera, vamos a ver cómo manipularla, es decir, cuál es la forma más sencilla de recibir dicha información.

La clase UT2004BotModuleController autoinicializa AnnotationListenerRegistrator. Ésto quiere decir que no será necesario manipular los listeners manualmente, es decir, tener que añadirlos, elminarlos, etc. además de reprogramarlos, saber cómo iterar para seleccionar el adecuado en cada momento, etc. En vez de eso, AnnotationListenerRegistrator permite registrar los listeners de forma muy sencilla y práctica, y él mismo se encargará de iterar entre todos los listeners registrados. Una vez registrado el listener, definimos una función que realice las acciones deseadas en cada caso.

Hay cinco diferentes tipos de listeners que podemos registrar. Los dos últimos son difíciles de usar, puesto que es difícil obtener el strig correspondiente al id específico del objeto, pero es necesario saber de su existencia, aunque todavía desconocemos cómo utilizarlos y si serán necesarios:

- EventListener: reacciona al evento que nosotros definamos por medio del campo eventClass. Por ejemplo:
  - Reg: *@EventListener(eventClass = Bumped.class)*, reacciona a eventos de la clase Bumped.
  - Proc: *protected void bumped(Bumped event)* {
- ObjectClassListener: reacciona a todos los eventos ocurridos a una clase de objeto concreta, definida por medio del campo objectClass. Por ejemplo:
  - Reg: *@ObjectClassListener (objectClass = Player.class)*, reacciona a eventos ocurridos a la clase Player.
  - Proc: *protected void player(Player object)* {
- ObjectClassEventListener: reacciona a eventos de una clase concreta (de entre los eventos asociados a objetos) que ocurren a objetos de una clase concreta.

- Ej: *@ObjectClassEventListener(eventClass = WorldObjectAppearedEvent.class, objectClass = Player.class)*, reacciona cuando un jugador cualquiera "aparece" en nuestro campo de visión.
- Proc: *protected void playerAppeared(WorldObjectAppearedEvent<Player> event)* {

- ObjectListener: similar a *ObjectClassListener* para un objeto concreto, es decir, un objeto con un id determinado dentro una clase concreta.
- ObjectEventListener: similar a *ObjectClassEventListener* para un objeto concreto, es decir, un evento de una clase concreta (de entre los eventos asociados a objetos) que ocurre a un objeto con un id determinado dentro una clase concreta.

Información que enviamos (Actions)

Como hemos dicho, nuestro bot será de la clase *UT2004BotModuleControler*, contenida en *cz.cuni.amis.pogamut.ut2004.bot.impl*. Para dar órdenes al mismo no usaremos directamente la interfaz *IAct*, es decir, pese a que somos libres de utilizar el método act (perteneciente como hemos dicho a la clase *UT2004BotModuleControler*), las acciones las enviaremos, de manera más sencilla e intuitiva, mediante el método body.

El método body está dividido en los siguiente métodos:

- Action getAction() Returns *cz.cuni.amis.pogamut.ut2004.bot.commands.Action* command module.
- Communication getCommunication() Returns *cz.cuni.amis.pogamut.ut2004.bot.commands.Communication* command module.
- ConfigureCommands getConfigureCommands() Returns *cz.cuni.amis.pogamut.ut2004.bot.commands.ConfigureC* command module.
- AdvancedLocomotion getLocomotion() Returns *cz.cuni.amis.pogamut.ut2004.bot.commands.AdvancedLocomotio* command module.
- AdvancedShooting getShooting() Returns *cz.cuni.amis.pogamut.ut2004.bot.commands.AdvancedShooting* command module.
- SimpleRayCasting getSimpleRayCasting() Returns *cz.cuni.amis.pogamut.ut2004.bot.commands.SimpleRayCastin* command module.

La clase UT2004BotModuleControler ofrece también los métodos shoot para acceder directamente a body.getShooting() y move para body.getLocomotion().

Para acceder a los mismos, debemos incluir las librerías correspondientes a cada clase contenida en Action, Comunication, ConfigureCommands, AdvancedLocomotion, AdvancedShooting y SimpleRayCasting. Para ello consultar la API.

A.4.2. Descripción de los eventos

En este apartado se trata de explicar de manera resumida los eventos más importantes que tendremos que utilizar durante la implementación de nuestro bot. Debemos tener en cuenta las siguientes consideraciones:

- Muchos de los eventos son utilizados para la conexión con el servidor y cosas por el estilo, lo cual ya está implementado y no necesitamos para implementar nuestro bot.

- Algunos de los eventos indican el principio y el final de la llegada de un lote síncrono de datos. (no sé si es necesario o está ya implementado en la clase que controle cada lote)
- Nuestra implementación está orientada al tipo de partida DeathMatch sin chat, por lo que no tendremos en cuenta eventos referentes a equipos, banderas, chat, etc.

Lotes síncronos

Comienzo y finalización de la transmisión de lotes síncronos

- HandShakeStart / HandShakeEnd
- ItemCategoryStart / ItemCategoryEnd
- ItemListStart / ItemListEnd
- MutatorListStart / MutatorListEnd
- MyInventoryStart / MyInventoryEnd
- PathListStart / PathListEnd
- PlayerListStart / PlayerListEnd

Eventos de objetos

Eventos especiales referentes a objetos. (ya explicados)

- WorldObjectEvent (superclase de los siguientes, es decir, cualquier evento de objeto)
- WorldObjectAppearedEvent
- WorldObjectDestroyedEvent
- WorldObjectDisappearedEvent
- WorldObjectFirstEncounteredEvent
- WorldObjectUpdatedEvent

Items / Inventary

- AdrenalineGained
- AddInventoryMsg
- ItemDescriptorObtained
- ItemPickedUp
- LostInventory
- WeaponUpdate
- Reachable

Puntos de Navegación

- NavPointNeighbourLink
- NavPointListStart / NavPointListEnd
- NavPointNeighbourLinkStart / NavPointNeighbourLinkEnd

Mapa

- MapFinished
- MapChange
- MapList
- MapListEnd
- MapListObtained
- MapListStart
- MapPointListObtained

Player

- PlayerDamaged
- PlayerInput
- PlayerJoinsGame
- PlayerKilled
- PlayerLeft
- PlayerListObtained
- PlayerScore
- ComboStarted

Mutator

- Mutator
- MutatorListObtained

Sonidos

- HearNoise
- HearPickup
- VolumeChanged

Mover

- MoverListEnd
- MoverListObtained
- MoverListStart

Dialog

- DialogCommand
- DialogFailed
- DialogOk

Vehículos

- EnteredVehicle
- LockedVehicle

Disparar

- ShootingStarted
- ShootingStopped

Bot

- BotDamaged
- BotFirstSpawned
- BotKilled
- Bumped
- ChangedWeapon
- FallEdge
- JumpPerformed
- Landed
- Spawn
- Thrown
- WallCollision
- ZoneChangedBot

Grabar

Respuesta a los comandos REC y STOPREC respectivamente.

- RecordingStarted
- RecordingEnded

Estado del juego

Se pone y se quita la pausa del juego.

- GameResumed
- GamePaused

Path

En la clase Path se encuentra la función getPath(). Una vez enviada esta petición, se nos devuelve un lote PathList cuyo comienzo y final está delimitado por PathListStart y PathListEnd respectivamente.

- Path
- PathList
- PathListStart / PathListEnd

Otros

- ObjectSelected
- BeginMessage
- EndMessage
- FactoryUsed
- FastTraceResponse
- GBEvent
- InitCommandRequest
- KeyEvent
- ListObtained
- ReadyCommandRequest
- TraceResponse
- Trigger
- WorldEventIdentityWrapper

Anexo B

BotPrize

A continuación se muestran las reglas de la competición BotPrize y los resultados obtenidos durante su edición de 2012.

B.1. Reglas de competición

B.1.1. El concurso

El objetivo de la competició es crear un bot para el videojuego Unreal Tournament 2004 que sea indistinguible de un jugador humano. Aquellos bots que consigan superar esta prueba compartirán un premio en efectivo de A$ 7.000 y también un viaje al estudio de 2K Australia en Canberra. Si ningún bot supera la pruba, se otorgará al ganador un premio menor de A$ 2,000 más un viaje al estudio. Como miembro del equipo ganador (pase la prueba o no) se invitará a visitar el estudio de 2K Australia, a expensas de 2K Australia, hasta por un monto de A$ 5.000, además del premio en metálico.

El concurso se llevará a cabo en la Universidad Edith Cowan en agosto de 2012, utilizando bots conectados remotamente por los competidores. Los resultados se darán a conocer en la *IEEE Conference on Computational Intelligence and Games*, en Granada, España, que tendrá lugar entre el 11 y el 14 de septiembre de 2012.

No hay ningún requisito para inscribirse o asistir a la conferencia (¡aunque sería bueno verte allí!), y no se espera que los competidores estén presentes en la propia competición.

El juego utilizado para el concurso se basa en una versión modificada del tipo de juego Death-Match para el FPS Unreal Tournament 2004. Esta versión modificada proporciona una interfaz basada en un socket (llamado Gamebots) que permite el control de los robots de un programa externo. Además, se hicieron varias modificaciones adicionales especiales para la competición:

El chat se desactivará (¡esto no es un concurso *chatbot*!) y algunos aspectos del juego se pueden modificar para faciltar la competencia.

B.1.2. Para participar

Los competidores deberán comunicar su intención de participar en el concurso a más tardar el miércoles 22 de agosto de 2012 por correo electrónico a los organizadores del concurso. La

competición se decidirá en una serie de sesiones que tendrán lugar el lunes 3 de septiembre de 2012.

Otras condiciones de entrada son:

- Nadie puede ingresar más de un bot (ya sea como individuo o como parte de un equipo), pero más de un equipo o individuo con la misma afiliación pueden entrar.
- Nadie asociado con 2K o con la organización de la competición puede entrar.
- Los participantes deben declarar que no tienen derechos de propiedad intelectual sobre el bot y que él y los componentes de su equipo cumplen con todas las licencias artísticas.
- Los participantes menores de 18 años deberán presentar una declaración por escrito del permiso de al menos un padre o tutor. Los participantes deben estar dispuestos a permitir que vídeos y/o *mpegs* de su participación en la competición puedan ser publicados y ser de dominio público.

B.1.3. Protocolo de las pruebas

La selección se realiza utilizando el mismo sistema de puntuación en el juego que se utilizó en 2011. El mod que se utilizará en el juicio se encuentra disponible en http://botprize.org/mod/modBP2012_3.zip.

El sistema se basa en una modificación del arma LinkGun. El modo principal se utiliza para marcar a un enemigo como BOT, mientras que el modo secundario es para etiquetar oponentes controlados por humanos. Durante el juego, cuando un jugador cree que ha identificado a un oponente como un BOT (o HUMANO), le dispara al oponente usando el modo principal (o secundaria) de la LinkGun. El disparo no tiene efecto sobre el oponente, pero el jugador verá una etiqueta "BOT" (o "HUMAN") unido al oponente para recordarles que los enemigos han sido juzgados. Si el jugador cambia de opinión, el oponente puede disparar usando el otro modo para revertir el fallo, las veces que desee.

La etiqueta final que tenga cada oponente en ronda final constituye el juicio de ese jugador a ese oponente. Al final de todas las rondas, la calificación de humanidad o de bot de cada jugador será el porcentaje de veces que se ha juzgado humana sobre todas las veces que se ha juzgado.

Se recomienda a los jueces que, además de juzgar, participen en el juego con normalidad. Como incentivo para los jueces para hacer esto, habrá premios para el mejor juez, y el juez con la puntuación combinada más alta durante los asaltos, y por el juez con la calificación de humanidad más alta.

B.1.4. To win

1. Para ganar el premio de A$ 7000, un bot debe alcanzar una calificación de humanidad igual o superior a las calificaciones promedio de humanidad de los jueces de la competencia.
2. Si más de un bot logra esta calificación, el premio se repartirá a partes iguales.
3. Si no se logra este que el bot alcance esta calificación, el premio de A$ 2000será compartido por los bots con la calificación de humanidad más alta.

B.2. Resultados BotPrize 2012

El porcentaje de humanidad se calcula dividiendo el número de veces que un jugador (bot o humano) fue juzgado como humano dividido por el número total de veces que se juzgó. Para este cálculo, sólo se tienen en cuanta los marcados hechos por humanos. A pesar de que a esto lo llamamos "humanidad", lo que realmente mide es cómo de parecido resulta un jugador al resto de jueces. Esto explica por qué es posible para los bots obtener una calificación más alta que la calificación de los verdaderos seres humanos - los bots son buenos engañando a los jueces.

Cada jugador se juzgó alrededor de 25 veces. Esto hace que sea poco probable que los altos índices de humanidad sea una casualidad. Por ejemplo, si un bot era realmente humano en un 35 %, entonces no sólo hay un 5 % de probabilidad de obtener más de un 50 % en la calificación. Si el bot era realmente sólo un 30 % humano, entonces este porcentaje cae a alrededor del 2 %.

Así como los seres humanos y los bots de la competención, algunos *epic bots*, los bots integrados en el juego, tomaron parte brevemente.

Nombre	**Equipo**	**Origen**	**Humanidad**
MirrorBot	Mihai Polceanu	1. ENIB CERV - Centre de Réalité Virtuelle, Brest, France 2. Ovidius University of Constanta - CeRVA Lab, Constanta, Romania	52.2 %
UTˆ2	Jacob Schrum, Igor Karpov, Risto Miikkulainen	University of Texas at Austin	51.9 %
ICE-CIG2012	Naoki Kusumoto, Takumi Sato, Seiji Murakami, Kenta Tsuji, Takahiro Yamamoto, Ruck Thawonmas	Ritsumeikan University, Japan	36.0 %
NeuroBot	Zafeirios Fountas, Murray Shanahan	Imperial College, London	26.1 %
GladiatorBot	David Holaň, Jakub Gemrot, Michal Bída	Charles University in Prague	21.7 %
AmisBot	Jan Dufek, Rudolf Kadlec	Charles University in Prague	16.0 %
average			34.2 %

Anexo C

ConsScale

ConsScale es una escala de inspiración biológica diseñada para evaluar implementaciones de Conciencia Artificial. La escala evalúa la presencia de funciones cognitivas asociadas con la conciencia.

ConsScale constituye un marco para la caracterización de las capacidades cognitivas de una criatura. ConsScale incluye la definición de una lista ordenada de niveles cognitivos organizados a lo largo de una trayectoria de desarrollo concreta. La organización particular de estos niveles está inspirada en la ontogenia y en la filogenia de la conciencia en organismos biológicos.

La suposición básica es que existen diferentes tipos de mentes y que éstas se pueden caracterizar en base a los criterios planteados en ConsScale. Usando esta escala, la caracterización y la evaluación del desarrollo de la conciencia se puede realizar usando tres herramientas relacionadas:

- El nivel conceptual de conciencia (Niveles ConsScale).
- El índice CQS (ConsScale Quantitative Score).
- La representación gráfica del perfil cognitivo.

Para evaluar el nivel de conciencia de un agente usando ConsScale, se necesitan identificar sus componentes arquitectónicos y probar la presencia de habilidades cognitivas. Usando esta información como entrada, se puede usar la escala para obtener tanto una medida cualitativa como una medida cuantitativa de la conciencia.

C.1. Niveles ConsScale

Cada nivel de la escala ConsScale se caracteriza en términos de criterios arquitectónicos y de comportamiento. El nivel específico que alcanza un agente indica su grado de desarrollo de la conciencia en términos de integración efectiva de funciones cognitivas clave.

La evaluación de un agente que se realiza utilizando la escala ConsScale se puede considerar también como una medida de la "Potencia Cognitiva" de un agente. Dado que las funciones cognitivas consideradas están asociadas a la conciencia, la medida obtenida es también una medida de conciencia. Cuanto más alto es el nivel obtenido, más se parece el comportamiento del agente al comportamiento de otras criaturas conscientes conocidas como los humanos.

La figura C.1 resume los principales niveles definidos en ConsScale. Estos hitos en el desarrollo son referencias asociadas a analogías biológicas. Es preciso aclarar que ConsScale no sólo es capaz

de proporcionar un nivel concreto, sino que establece de forma precisa el grado de desarrollo (que podría estar en cualquier punto entre dos niveles canónicos).

En ConsScale se definen 13 niveles diferentes de conciencia funcional. Cada nivel se caracteriza en base a criterios arquitectónicos y de comportamiento. A continuación se proporciona una descripción de las principales características de cada nivel, la definición de los componentes arquitectónicos abstractos y las habilidades cognitivas específicas asociadas a los niveles de ConsScale.

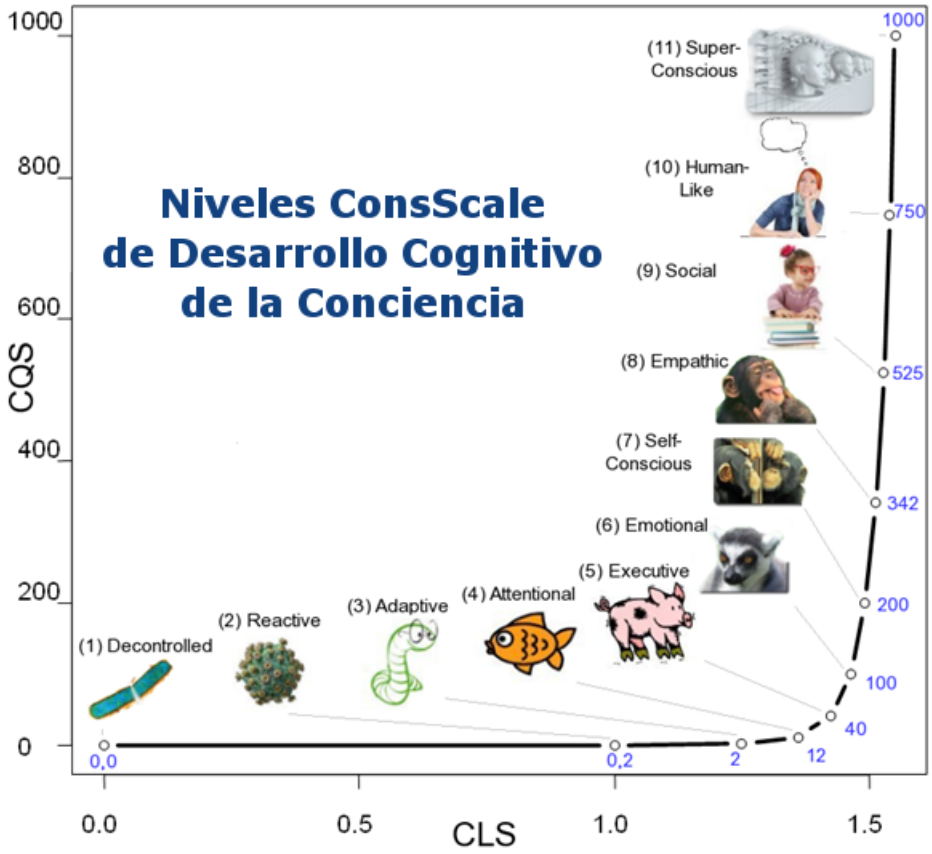

Figura C.1: Niveles ConsScale

Nivel -1. Sin Cuerpo Definido (Disembodied)

Este es un nivel inicial de referencia que se corresponde con implementaciones muy simples que ni siquiera tienen una definición clara de las fronteras que separan el agente del entorno que lo rodea. En otras palabras, este nivel se refiere a implementaciones que no pueden ser consideradas como agentes y se pueden confundir fácilmente con el resto del entorno. Desde el punto de vista del desarrollo cognitivo propuesto en ConsScale, las entidades de nivel -1 podrían considerarse como "proto-agentes". Aunque este nivel no se use de forma práctica en la evaluación de posibles implementaciones, su definición sirve para poner de manifiesto la importancia del cuerpo (B) como requisito básico para la descripción de un agente situado. No hay habilidades cognitivas características de este nivel. Una analogía tomada del mundo biológico para los individuos de este nivel sería la consideración de un aminoácido como parte de una proteína. El resto de la escala consiste en un conjunto de doce niveles (del 0 al 11, ambos inclusive), en la que los niveles más altos incluyen

como parte de su propia definición a todos los niveles inferiores. Usando estos niveles, se puede caracterizar cualitativamente el desarrollo cognitivo de un agente artificial.

Nivel 0. Aislado (Isolated)

Al igual que el nivel -1, el nivel 0 es un nivel de referencia conceptual que permite resaltar la importancia de la interacción con el entorno (situatedness), cuya base se encuentra en la existencia de un cuerpo. En este nivel, aunque hay una clara distinción entre la propia implementación (su cuerpo) y el entorno, hay una falta total de procesamiento autónomo y no existen sistemas sensoriales ni motores. Por lo tanto, una entidad de nivel 0 consiste únicamente en un cuerpo inerte que no presenta ninguna funcionalidad ni interacción activa con el medio (excepto la inevitablemente provocada por las propiedades físicas de la entidad). En este nivel tampoco hay habilidades cognitivas características. Un cromosoma aislado podría considerarse como una analogía válida para este nivel.

Nivel 1. Pre-funcional (Decontrolled)

Este nivel se refiere a aquellas implementaciones en las que los subsistemas correspondientes a sensores y actuadores están presentes, pero o bien no funcionan o no existe una relación funcional entre ellos. Como ni la adquisición de información ni la generación de acciones funcionan o no están relacionadas, todavía no se pueden definir habilidades cognitivas características a este nivel. Una bacteria muerta podría ser una analogía plausible tomada del mundo biológico.

Nivel 2. Reactivo (Reactive)

En este nivel, tanto los subsistemas de sensores como los actuadores son funcionales y están relacionados entre ellos mediante funciones predefinidas. El comportamiento de estos agentes está caracterizado por la producción de respuestas reactivas fijas en base a los datos de entrada capturados por los sensores. Por lo tanto, la única capacidad cognitiva característica de este nivel es la de un agente situado que responde al entorno siempre con los mismos reflejos. Una analogía biológica para este nivel podría ser un virus. El nivel 2 de ConsScale se corresponde con un agente reactivo clásico que carece de memoria explícita y tampoco dispone de mecanismos de aprendizaje. Es a partir del nivel 2, cuando los agentes empiezan a utilizar el propio entorno que les rodea como medio para cerrar el bucle de retroalimentación entre la acción y la percepción. Por lo tanto, todos los agentes de nivel 2 o superior pueden considerarse agentes situados.

Aunque la escala se centra principalmente en la evaluación de agentes individuales, es importante destacar que incluso en el nivel 2 pueden existir procesos adicionales de aprendizaje y adaptación en el plano evolutivo (suponiendo que los agentes sean capaces de replicarse, mutar y evolucionar). Por ejemplo, aunque las reglas de control reactivas son fijas en un único individuo de nivel 2, podría aparecer un proceso de adaptación de las respuestas reactivas a lo largo de sucesivas generaciones en una población de agentes de nivel 2.

Nivel 3. Adaptativo (Adaptive)

A este nivel las acciones del agente se generan dinámicamente en función tanto de la memoria como de la información que se obtiene del entorno por medio de los sensores. Las habilidades cognitivas características de este nivel son la capacidad básica para aprender nuevos reflejos y el uso de sensores propioceptivos para la generación de comportamientos básicos como los de orientación y posicionamiento. La lombriz de tierra sería una analogía biológica ilustrativa para este nivel.

Un agente de nivel 3 se corresponde con la forma más simple de un agente deliberativo. En este nivel, el estado interno del agente se mantiene en un sistema de memoria. La coordinación sensoriomotora (R) se genera en función de la información percibida y la información recordada. En este nivel, también se considera la presencia de sensores propioceptivos, aunque esto por sí solo no es suficiente para generar autoconciencia. Los mecanismos de percepción propioceptiva permiten que parte del estado interno del agente forme parte de la entrada de la función de coordinación sensoriomotora. Los agentes de nivel 3 también tienen mecanismos de aprendizaje que les permiten descubrir nuevos comportamientos reactivos. Es decir, la respuesta a un determinado estado del entorno no es fija, sino que es una función de la información adquirida por S y el estado interno del agente (M). El nivel 3 también se puede ver como una evolución del nivel 2 en la que ha aparecido la capacidad de aprender nuevos reflejos.

Nivel 4. Atencional (Attentional)

A este nivel el comportamiento del agente está modulado por la influencia que ejerce un mecanismo de atención. La atención selecciona contenidos específicos del repertorio total de contenidos disponibles a través de los sensores y de la memoria. Además, los contenidos seleccionados se evalúan positiva o negativamente, constituyendo esto la semilla de las emociones. Las capacidades cognitivas típicas de un agente de nivel 4 permiten la producción de comportamientos de ataque y de escape (o de acercamiento y alejamiento). Los peces podrían constituir una analogía biológica para este nivel.

Gracias al mecanismo de atención los procesos de aprendizaje se pueden dirigir explícitamente hacia determinados objetos o sucesos. Además, los procesos de aprendizaje implícito, como la adquisición de nuevos reflejos (característica del nivel anterior), también se dan al mismo tiempo. Los agentes de nivel 4 son capaces de dirigir la atención (usando el componente Att) a un subconjunto seleccionado de los contenidos percibidos del entorno (Ei), mientras que otras variables ambientales, que son adquiridas en S, no son procesadas explícitamente en R. Los objetos o sucesos seleccionados por el foco de atención son evaluados automáticamente en base a las metas del propio agente. Esto permite que las respuestas subsiguientes del agente hacia esos estímulos se adapten (este mecanismo se puede considerar la base para el desarrollo de las emociones).

Los agentes atencionales son capaces de desarrollar comportamientos dirigidos, como los de ataque o escape, y también son capaces de implementar mecanismos de aprendizaje por prueba y error. La habilidad de prestar atención hacia estímulos específicos permite la formación de comportamientos dirigidos. Por ejemplo, un agente puede desarrollar comportamientos claramente relacionados con objetivos específicos, como la persecución o la huída. Adicionalmente, los agentes de nivel 4 tienen mecanismos primitivos para las emociones ya que los objetos a los que se presta atención son evaluados elementalmente como positivos o negativos. Una emoción positiva desencadena un comportamiento de acercamiento o un vínculo hacia el objeto seleccionado. Por el contrario, una emoción negativa desencadena un comportamiento de alejamiento y refuerzo de las fronteras entre el agente y el objeto seleccionado. Adicionalmente, aparece en este nivel una nueva relación entre la memoria y las emociones. Como se ha demostrado con los organismos biológicos, las emociones influyen en gran medida en la selección de los contenidos que se almacenan en memoria. En resumen, un agente de nivel 4 puede considerarse una evolución de un agente de nivel 3 en el que ha aparecido la capacidad de atención.

Nivel 5. Ejecutivo (Executive)

Los agentes de este nivel son capaces de intercalar múltiples metas ya que son capaces de almacenar en memoria distintos conjuntos de trabajo. Las habilidades cognitivas características de este nivel son el cambio de contexto (habilidad para pasar de una tarea a otra) y el aprendizaje

emocional básico. La capacidad de cambio de contexto implica que el agente puede suspender la realización de una tarea dada para retomarla más tarde, estableciendo prioridades entre todas las tareas pendientes. El agente puede perseguir múltiples metas, asignando más tiempo y esfuerzo a aquellas que reportan mayores beneficios emocionales. Los mamíferos cuadrúpedos son una analogía biológica ilustrativa para este nivel.

Un agente de nivel 5 se caracteriza por una capacidad de razonamiento más compleja y una representación más rica del estado interno que permite la implementación de mecanismos de cambio de contexto. La consecución de múltiples metas se consigue gracias a un mecanismo de coordinación que es capaz de mover el foco de atención de forma efectiva de una tarea a otra. Un agente de nivel 5 también está dotado de un mecanismo que permite evaluar el propio desempeño en la consecución de las metas pendientes. Se trata del mecanismo de auto-evaluación del propio estado (SsA), que puede ser identificado como las emociones.

La presencia de emociones asociadas a objetos, sucesos, y ahora también a las propias acciones del agente, permite el desarrollo de procesos de aprendizaje por refuerzo. Un agente de nivel 5 es aquel que exhibe capacidades de cambio de contexto y aprendizaje emocional básico (aprendizaje por refuerzo). Otras características de un agente ejecutivo son la capacidad de planificación avanzada y la aplicación de las emociones al cambio de contexto: el agente tiende a asignar más tiempo y esfuerzo a aquellas tareas que resultan más gratificantes para él. En resumen, un agente de nivel 5 se puede considerar como una evolución de un agente de nivel 4 en la que ha aparecido la capacidad de perseguir múltiples metas y intercalar la realización de múltiples tareas.

Nivel 6. Emocional (Emotional)

Este nivel se caracteriza por la capacidad de desarrollar el estadio 1 de la Teoría de la Mente o TdM: "Yo sé". La TdM es la capacidad de atribuir estados mentales a uno mismo y a otros sujetos. En el nivel 6 de ConsScale los sentimientos aparecen como representaciones de los cambios que experimenta el organismo debido a las emociones. El sentido de "yo sé" aparece en el agente gracias a la representación de las relaciones existentes entre las emociones y los estados del organismo. La habilidad cognitiva característica del nivel 6 es la capacidad de aprendizaje emocional. El agente generaliza las lecciones aprendidas y las aplica a su comportamiento global. Además, las emociones se asignan también al propio yo y al proceso de auto-monitorización, produciendo una auto-evaluación continua que da lugar al "yo sé". Los monos son una analogía biológica plausible para el nivel 6.

El nivel 6 es el primer nivel de la escala ConsScale en el que se puede considerar que un agente es hasta cierto punto consciente (pero no autoconsciente). La principal característica de este nivel es, como se ha mencionado anteriormente, el desarrollo del estadio 1 de la TdM. Aparecen las emociones complejas, que se construyen como combinaciones de las emociones básicas presentes en los niveles anteriores. Estas emociones no evalúan sólo estímulos externos o el propio estado interno, sino que se generan emociones de fondo que evalúan la relación del agente con su entorno.

Como en este nivel los agentes cuentan con una representación precisa de su estructura corporal y sus capacidades físicas, es más fácil establecer una separación entre el yo y el resto del mundo. Por ejemplo, un robot antropomórfico de nivel 6 actualizará su modelo del yo al descubrir la diferencia existente entre tocar con la mano un objeto extraño y tocar su propio cuerpo. En el segundo caso, se produce una notificación de los sensores de contacto de la zona del cuerpo que toca la mano al mismo tiempo que se ejecuta la acción. Esta capacidad permite establecer una distinción clara entre los objetos que se pueden controlar directamente (el propio cuerpo) y los que no.

La sensación correspondiente al "yo sé" aparece en el agente gracias a la representación que se genera de la respuesta del organismo a las emociones. Existe un modelo implícito del yo que permite asociar las emociones con el desempeño del propio agente. Usando estas representaciones el agente es capaz de generalizar en sus procesos de aprendizaje. Mientras que un agente de nivel

5 sólo aprende las reglas específicas de una tarea y adaptar su comportamiento consecuentemente, un agente de nivel 6 es capaz de usar las emociones complejas para aprender lecciones básicas que se pueden generalizar a todo su comportamiento, independientemente de la tarea que se realice (generalización de lecciones aprendidas). Por ejemplo, desarrollar un miedo a manejar cargas de más de 500 kilogramos podría hacer que un hipotético robot de carga de nivel 6 evitase tanto las tareas conocidas como nuevas tareas que involucren el levantamiento y transporte de ese tipo de cargas. Es decir, el agente habría aprendido la lección de que en general (él) no es capaz de manejar cargas tan pesadas.

Nivel 7. Autoconsciente (Self-Conscious)

La autoconciencia se adquiere cuando el agente es capaz de desarrollar el estadio 2 de desarrollo de la TdM: "yo sé que yo sé". La presencia de un modelo explícito del yo en el agente hace posible el auto-reconocimiento. De hecho, los mecanismos de aprendizaje ahora pueden operar en el dominio del propio futuro anticipado. El agente puede planear acerca de sí mismo (ya que el propio agente puede ser parte del plan) y luego aprender si el plan fue eficiente para él o no. Aprender a usar herramientas es una capacidad cognitiva característica de este nivel, ya que ser actor en el plan es un factor clave para el uso de herramientas. Los humanos de 18 meses de edad son una analogía biológica plausible para este nivel.

El nivel 7 de ConsScale se corresponde con la aparición de la autoconciencia. En este nivel el agente es capaz de desarrollar pensamientos de orden superior, es decir, pensamientos sobre pensamientos, y más específicamente pensamientos sobre uno mismo. Consecuentemente, los agentes de nivel 7 se encuentran en el estadio 2 del desarrollo de la TdM: "yo sé que yo sé". Esto requiere la presencia de un modelo explícito del yo, que a su vez permite la planificación avanzada incluyendo al propio yo en los planes.

En el nivel 7 los mecanismos de aprendizaje operan también en el dominio del futuro anticipado. El agente puede realizar planes sobre sí mismo, y después de ejecutarlos, evaluar si el plan confeccionado fue beneficioso para el agente o no. Es más, un agente de nivel 7 tiene capacidad de imaginación. Es decir, puede planificar en base a los resultados de simulaciones internas, que son capaces de predecir los resultados de las posibles acciones del agente.

Al existir un símbolo explícito para el yo, el agente de nivel 7 es capaz de reconocerse a sí mismo. Por lo tanto, el test de comportamiento característico para este nivel sería la prueba del espejo (ver Apartado 2.3.2). El comportamiento de los agentes de este nivel también se caracteriza por la capacidad de usar herramientas.

Nivel 8. Empático (Empathic)

La intersubjetividad es la característica principal de este nivel, en el que el agente no sólo está dotado de un modelo interno mejorado que incluye el yo, sino que también tiene la habilidad de modelar a otros como yos intencionales. El desarrollo del estadio 3 de TdM, "yo sé que tú sabes", hace posible los comportamientos sociales. Los chimpancés son una analogía biológica ilustrativa para este nivel.

En el nivel 8 las representaciones internas del agente se enriquecen al contemplarse la intersubjetividad. Además del modelo del yo, característico del nivel anterior, el individuo mantiene de forma análoga modelos actualizados de otros individuos (otros). Es decir, el agente asigna a otros individuos un modelo de subjetividad (se aplica un modelo del yo también a otros individuos). Esta capacidad es la base de la interacción social compleja.

Tanto la intersubjetividad como la capacidad de mantener un esquema corporal actualizado (la cual estaba presente ya en el nivel 6 de ConsScale), son necesarias para el aprendizaje del

uso de herramientas mediante imitación e incluso para la fabricación de nuevas herramientas. Los agentes de nivel 8 también se caracterizan por ser capaces de colaborar con otros agentes en la persecución de una meta común. Este tipo de agentes pueden desarrollar planes que tienen en cuenta la dimensión social. Es decir, la relación existente entre el modelo del yo y los modelos de otros individuos subjetivos. La necesidad de estas habilidades cognitivas se ha puesto de manifiesto en el diseño de agentes BDI que sean capaces de colaborar entre ellos y/o con humanos.

Nivel 9. Social (Social)

En este nivel el modelo interno de otros yos se perfecciona gracias al desarrollo total de la capacidad de TdM, "yo sé que tú sabes que yo sé". Esto significa que el comportamiento característico de este nivel viene definido por el desarrollo de estrategias Maquiavélicas sofisticadas (o inteligencia social), entre las que se incluyen comportamientos sociales como la mentira, la astucia o el liderazgo. Además, los agentes de nivel 9 cuentan con capacidades lingüísticas y comunicación precisa. Los agentes de este nivel serían capaces de desarrollar una cultura propia. Los humanos de 4 años de edad son la analogía biológica para este nivel.

En el nivel 9, la TdM está totalmente desarrollada, por lo tanto los agentes están fuertemente influidos por su entorno social. Además, el desarrollo de una cultura proporciona nuevas posibilidades de aprendizaje. Un agente social A (en términos de la escala ConsScale) sería consciente de que otro agente B podría conocer las creencias, los deseos y las intenciones de A. Este tipo de agentes también se caracterizan por sus capacidades avanzadas de comunicación, que junto con el desarrollo de la TdM, les hace capaces de, por ejemplo, contar mentiras intencionadamente. Existen modelos matemáticos de la dinámica de la inteligencia Maquiavélica que se podrían usar potencialmente para descubrir este tipo de comportamientos en agentes artificiales.

Nivel 10. Androide (Human-Like)

Tal y como indica el nombre de este nivel, la analogía biológica correspondiente es el ser humano adulto. Las características de este nivel son la capacidad para formar una cultura compleja y la comunicación verbal precisa. Esto implica la capacidad de uso de herramientas externas complejas para el aprendizaje. La fluidez entre la inteligencia social y la inteligencia técnica permite la extensión del conocimiento usando medios externos (como la comunicación escrita). Los avances tecnológicos son posibles en este nivel. Los agentes de nivel 10, al igual que los humanos, son capaces de modificar su entorno de forma extrema.

El nivel 10 representa a la clase de agentes dotados con un nivel de conciencia equiparable al humano. Esto implica que los grupos de este tipo de agentes pueden formar una cultura compleja o formar parte de la cultura humana. El Test de Turing, en cualquiera de sus formas o variantes, es una prueba obvia para los agentes de este nivel.

Nivel 11. Super-Consciente (Super-Conscious)

Este último nivel está caracterizado por la habilidad de sincronizar y coordinar varios flujos de conciencia en el mismo yo físico. No hay ejemplos conocidos de esta habilidad en el mundo biológico.

Un agente super-consciente es aquel capaz de manejar internamente varios flujos de conciencia, coordinando al mismo tiempo un único cuerpo y su correspondiente sistema de atención concurrente. Los agentes de este tipo requieren un mecanismo de coordinación entre los distintos flujos de conciencia y un mecanismo de acceso sincronizado a los recursos físicos. Un agente de este tipo

sería capaz, por ejemplo, de mantener varias conversaciones conscientes simultáneas utilizando diferentes líneas de comunicación. Además, podría usar la información obtenida de una conversación en cualquiera de las otras de forma prácticamente inmediata.

C.2. Proceso de Evaluación de ConsScale

Este proceso de evaluación está orientado a agentes implementados y proporciona una medida precisa del nivel de desarrollo cognitivo. Sin embargo, requiere mucho más tiempo y esfuerzo en su realización.

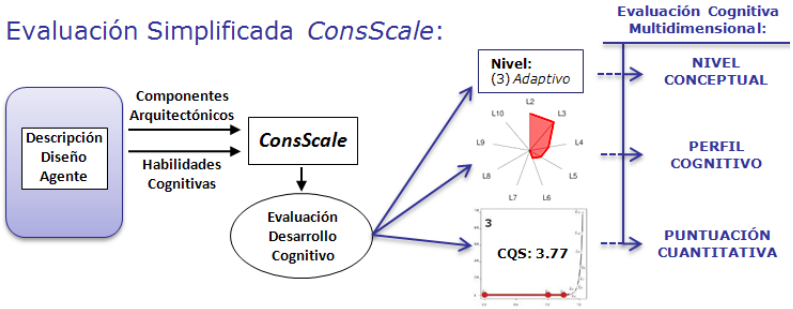

Figura C.2: Proceso de evaluación ConsScale (PSE)

La realización de un Proceso de Evaluación Estándar (PEE) requiere disponer del agente que se quiere evaluar y también una definición de pruebas adaptada al dominio de problema correspondiente, de manera que habitualmente se utiliza el Proceso de Evaluación Estándar (PSE). Como muestra la figura anteior, la evaluación se basa en los componentes arquitectónicos del agente y sus capacidades cognitivas. Los componentes arquitectónicos se pueden identificar a través de la inspección interna de la implementación. Las habilidades cognitivas presentes en el agente se pueden evaluar gracias a la definición y ejecución de pruebas de comportamiento específicas adaptadas para el dominio de problema seleccionado.

Las habilidades cognitivas descritas para cada nivel de la escala ConsScale se refieren a capacidades genéricas. Por lo tanto, la lista de habilidades cognitivas no se pueden usar directamente para realizar una evaluación estándar. Se requiere de un proceso de instanciación para poder aplicar la escala a un dominio de aplicación concreto. El proceso de instanciación consiste básicamente en diseñar pruebas conductuales, específicas para el dominio correspondiente, que permitan determinar la presencia de las habilidades cognitivas consideradas en cada nivel de la escala.

Una vez que se dispone de la lista de componentes arquitectónicos y habilidades cognitivas presentes en el agente se puede aplicar la escala para obtener el nivel nominal de conciencia, el perfil cognitivo y el índice CQS (usando por ejemplo la Calculadora ConsScale).

Anexo D

Soar

Este manual ha sido extraído de ***The Soar User's Manual Version 9.3.1*** (soar.googlecode.com/files/SoarManu

This chapter describes the Soar architecture. It covers all aspects of Soar except for the specic syntax of Soar's memories and descriptions of the Soar user-interface commands.

This chapter gives an abstract description of Soar. It starts by giving an overview of Soar and then goes into more detail for each of Soar's main memories (working memory, production memory, and preference memory) and processes (the decisión procedure, learning, and input and output).

D.1. An Overview of Soar

The design of Soar is based on the hypothesis that all deliberate goal -oriented behavior can be cast as the selection and application of operators to a state. A state is a representation of the current problem-solving situation; an operator transforms a state (makes changes to the representation); and a goal is a desired outcome of the problem-solving activity.

As Soar runs, it is continually trying to apply the current operator and select the next operator (a state can have only one operator at a time), until the goal has been achieved. The selection and application of operators is illustrated in Figure D.1.

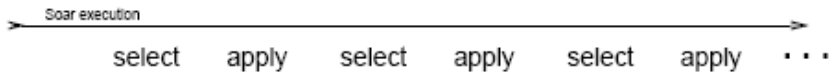

Figura D.1: Soar is continually trying to select and apply operators.

Soar has separate memories (and dierent representations) for descriptions of its current situation and its long-term knowledge. In Soar, the current situation, including data from sensors, results of intermediate inferences, active goals, and active operators is held in working memory. Working memory is organized as objects. Objects are described in terms of their attributes; the values of the attributes may correspond to sub-objects, so the description of the state can have a hierarchical organization. (This need not be a strict hierarchy; for example, there's nothing to prevent two objects from being \substructure" of each other.)

The long-term knowledge, which species how to respond to dierent situations in working memory, can be thought of as the program for Soar. The Soar architecture cannot solve any problems

without the addition of long-term knowledge. (Note the distinction between the \Soar architecture" and the \Soar program": The former refers to the system described in this manual, common to all users, and the latter refers to knowledge added to the architecture.)

A Soar program contains the knowledge to be used for solving a specic task (or set of tasks), including information about how to select and apply operators to transform the states of the problem, and a means of recognizing that the goal has been achieved.

D.1.1. Problem-Solving Functions in Soar

All of Soar's long-term knowledge is organized around the functions of operator selection and operator application, which are organized into four distinct types of knowledge:

- **Knowledge to select an operator :**
  1. *Operator Proposal*: Knowledge that an operator is appropriate for the current situation.
  2. *Operator Comparison*: Knowledge to compare candidate operators.
  3. *Operator Selection*: Knowledge to select a single operator, based on the comparisons.
- **Knowledge to apply an operator** :

  4.Operator Application: Knowledge of how a specic operator modies the state.

In addition, there is a fth type of knowledge in Soar that is indirectly connected to both operator selection and operator application:

- 5. Knowledge of monotonic inferences that can be made about the state (*state elaboration*).

State elaborations indirectly aect operator selection and application by creating new descriptions of the current situation that can cue the selection and application of operators.

These problem-solving functions are the primitives for generating behavior in Soar. Four of the functions require retrieving long-term knowledge that is relevant to the current situation: elaborating the state, proposing candidate operators, comparing the candidates, and applying the operator by modifying the state. These functions are driven by the knowledge encoded in a Soar program. Soar represents that knowledge as production rules. Production rules are similar to \if-then" statements in conventional programming languages. (For example, a production might say something like \if there are two blocks on the table, then suggest an operator to move one block ontop of the other block"). The \if" part of the production is called its conditions and the \then" part of the production is called its actions. When the conditions are met in the current situation as dened by working memory, the production is matched and it will re, which means that its actions are executed, making changes to working memory.

The other function, selecting the current operator, involves making a decisión once sucient knowledge has been retrieved. This is performed by Soar's decisión procedure, which is a xed procedure that interprets preferences that have been created by the retrieval functions. The knowledge-retrieval and decision-making functions combine to form Soar's decisión cycle.

When the knowledge to perform the problem-solving functions is not directly available in productions, Soar is unable to make progress and reaches an impasse. There are three types of possible impasses in Soar:

1. An operator cannot be selected because none are proposed.

2. An operator cannot be selected because multiple operators are proposed and the comparisons are insucient to determine which one should be selected.

3. An operator has been selected, but there is insucient knowledge to apply it.

In response to an impasse, the Soar architecture creates a substate in which operators can be selected and applied to generate or deliberately retrieve the knowledge that was not directly available; the goal in the substate is to resolve the impasse. For example, in a substate, a Soar program may do a lookahead search to compare candidate operators if comparison knowledge is not directly available. Impasses and substates are described in more detail in Section 2.6. SECCION

D.1.2. An Example Task: The Blocks-World

We will use a task called the blocks-world as an example throughout this manual. In the blocks-world task, the initial state has three blocks named A, B, and C on a table; the operators move one block at a time to another location (on top of another block or onto the table); and the goal is to build a tower with A on top, B in the middle, and C on the bottom. The initial state and the goal are illustrated in Figure D.2

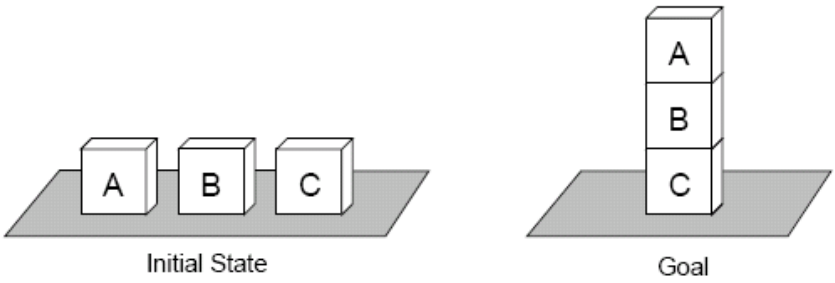

Figura D.2: The initial state and goal of the “blocks-world” task.

The operators in this task move a single block from its current location to a new location; each operator is represented with the following information:

- the name of the block being moved
- the current location of the block (the \thing” it is on top of)
- the destination of the block (the \thing” it will be on top of)

The goal in this task is to stack the blocks so that C is on the table, with block B on block C, and block A on top of block B.

D.1.3. Representation of States, Operators, and Goals

The initial state in our blocks-world task | before any operators have been proposed or selected | is illustrated in Figure D.3.

A state can have only one operator at a time, and the operator is represented as substructure of the state. A state may also have as substructure a number of potential operators that are in consideration; however, these suggested operators should not be confused with the current operator.

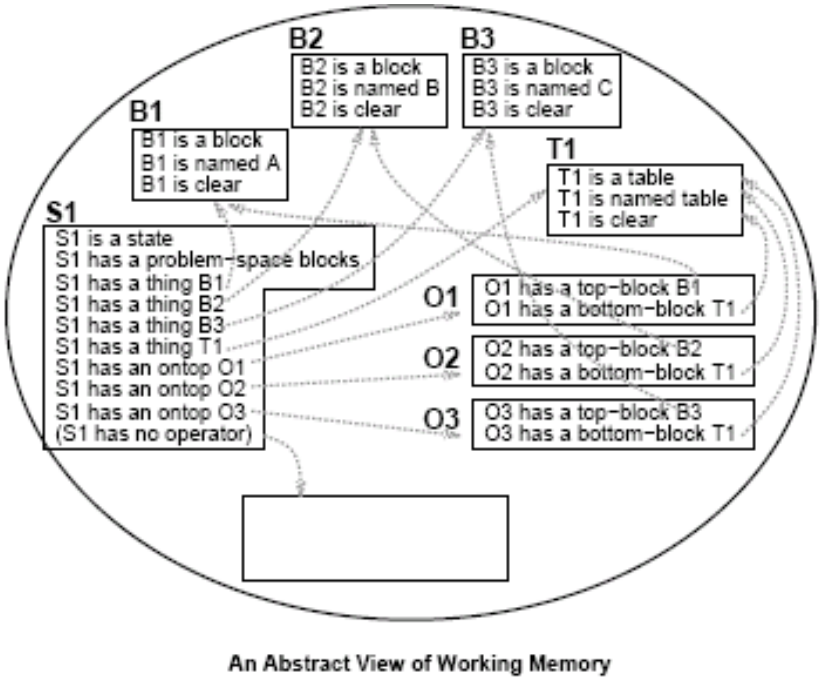

Figura D.3: An abstract illustration of the initial state of the blocks world as working memory objects. At this stage of problem solving, no operators have been proposed or selected.

Figure D.4 illustrates working memory after the first operator has been selected. There are six operators proposed, and only one of these is actually selected.

Goals are either represented explicitly as substructure of the state with general rules that recognize when the goal is achieved, or are implicitly represented in the Soar program by goal-specic rules that test the state for specic features and recognize when the goal is achieved. The point is that sometimes a description of the goal will be available in the state for focusing the problem solving, whereas other times it may not. Although representing a goal explicitly has many advantages, some goals are dicult to explicitly represent on the state.

The goal in our blocks-world task is represented implicitly in the Soar program. A single production rule monitors the state for completion of the goal and halts Soar when the goal is achieved.

D.1.4. Proposing candidate operators

As a first step in selecting an operator, one or more candidate operators are proposed. Operators are proposed by rules that test features of the current state. When the blocksworld task is run, the Soar program will propose six distinct (but similar) operators for the initial state as illustrated in Figure These operators correspond to the six dierent actions that are possible given the initial state.

D.1.5. Comparing candidate operators: Preferences

The second step Soar takes in selecting an operator is to evaluate or compare the candidate operators. In Soar, this is done via rules that test the proposed operators and the current state,

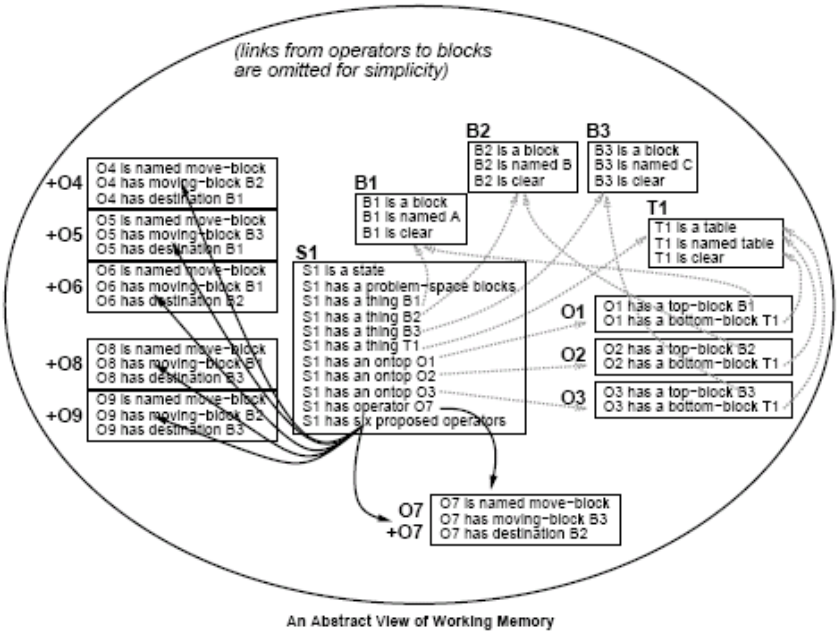

Figura D.4: An abstract illustration of working memory in the blocks world after the rst operator has been selected.

and then create preferences. Preferences assert the relative or absolute merits of the candidate operators. For example, a preference may say that operator A is a "better" choice than operator B at this particular time, or a preference may say that operator A is the "best" thing to do at this particular time.

D.1.6. Selecting a single operator

Soar attempts to select a single operator based on the preferences available for the candidate operators. There are four dierent situations that may arise:

1. The available preferences unambiguously prefer a single operator.
2. The available preferences suggest multiple operators, and prefer a subset that can be selected from randomly.
3. The available preferences suggest multiple operators,but neither case 1 or 2 above hold.
4. The available preferences do not suggest any operators.

In the first case, the preferred operator is selected. In the second case, one of the subset is selected randomly. In the third and fourth cases, Soar has reached an "impasse" in problem solving, and a new substate is created. Impasses are discussed in Section SECCION In our blocks-world example, the second case holds, and Soar can select one of the operators randomly.

D.1.7. Applying the operator

An operator applies by making changes to the state; the specic changes that are appropriate depend on the operator and the current state.

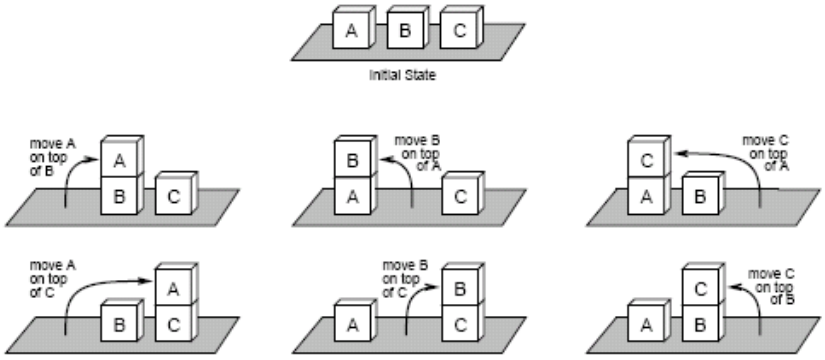

Figura D.5: The six operators proposed for the initial state of the blocks world each move one block to a new location.

There are two primary approaches to modifying the state: indirect and direct. Indirect changes are used in Soar programs that interact with an external environment: The Soar program sends motor commands to the external environment and monitors the external environment for changes. The changes are re ected in an updated state description, garnered from sensors. Soar may also make direct changes to the state; these correspond to Soar doing problem solving "in its head". Soar programs that do not interact with an external environment can make only direct changes to the state.

Internal and external problem solving should not be viewed as mutually exclusive activities in Soar. Soar programs that interact with an external environment will generally have operators that make direct and indirect changes to the state: The motor command is represented as substructure of the state and it is a command to the environment. Also, a Soar program may maintain an internal model of how it expects an external operator will modify the world; if so, the operator must update the internal model (which is substructure of the state).

When Soar is doing internal problem solving, it must know how to modify the state descriptions appropriately when an operator is being applied. If it is solving the problem in an external environment, it must know what possible motor commands it can issue in order to affect its environment.

The example blocks-world task described here does not interact with an external environment. Therefore, the Soar program directly makes changes to the state when operators are applied. There are four changes that may need to be made when a block is moved in our task:

1. The block that is being moved is no longer where it was (it is no longer "on top" of the same thing).

2. The block that is being moved is now in a new location (it is "on top" of a new thing).

3. The place that the block used to be is now clear.

4. The place that the block is moving to is no longer clear | unless it is the table, which is always considered "clear".

The blocks-world task could also be implemented using an external simulator. In this case, the Soar program does not update all the "on top" and "clear" relations; the updated state description comes from the simulator.

D.1.8. Making inferences about the state

Making monotonic inferences about the state is the other role that Soar long-term knowledge may fulll. Such elaboration knowledge can simplify the encoding of operators because entailments of a set of core features of a state do not have to be explicitly included in application of the operator. In Soar, these inferences will be automatically retracted when the situation changes such that the inference no longer holds.

For instance, our example blocks-world task uses an elaboration to keep track of whether or not a block is "clear". The elaboration tests for the absence of a block that is "on top" of a particular block; if there is no such "on top", the block is "clear". When an operator application creates a new "on top", the corresponding elaboration retracts, and the block is no longer "clear".

D.2. Working memory: The Current Situation

Soar represents the current problem-solving situation in its working memory. Thus, working memory holds the current state and operator and is Soar's "short-term" knowledge, re ecting the current knowledge of the world and the status in problem solving.

Working memory contains elements called working memory elements, or WME's for short. Each WME contains a very specic piece of information; Each WME describes a dierent attribute of the object, for example, its name or type or location; each attribute has a value associated with it, for example, the name is A, the type is block, and the position is on the table. Therefore, each WME is an identier-attribute-value triple, and all WME's with the same identier are part of the same object.

WME's are also often called augmentations because they "augment" the object, providing more detail about it. While these two terms are somewhat redundant, WME is a term that is used more often to refer to the contents of working memory, while augmentation is a term that is used more often to refer to the description of an object.

The elements in working memory arise from one of four sources:

1. The actions of productions create most working memory elements.
2. The decisión procedure automatically creates some special state augmentations (type, superstate, impasse, ...) whenever a state is created. States are created during initialization (the rst state) or because of an impasse (a substate).
3. The decisión procedure creates the operator augmentation of the state based on preferences. This records the selection of the current operator.
4. External I/O systems create working memory elements on the input-link for sensory data.

The elements in working memory are removed in six dierent ways:

1. The decisión procedure automatically removes all state augmentations it creates when the impasse that led to their creation is resolved.
2. The decisión procedure removes the operator augmentation of the state when that operator is no longer selected as the current operator.
3. Production actions that use reject preferences remove working memory elements that were created by other productions.

4. The architecture automatically removes i-supported WMEs when the productions that created them no longer match.

5. The I/O system removes sensory data from the input-link when it is no longer valid.

6. The architecture automatically removes WME's that are no longer linked to a state (because some other WME has been removed).

Preferences are held in a separate preference memory where they cannot be tested by productions; however, acceptable preferences are held in both preference memory and in working memory. By making the acceptable preferences available in working memory, the acceptable preferences can be tested for in productions allowing the candidates operators to be compared before they are selected.

D.3. Production Memory: Long-term Knowledge

Soar represents long-term knowledge as productions that are stored in production memory. Each production has a set of conditions and a set of actions. If the conditions of a production match working memory, the production res, and the actions are performed.

D.3.1. The structure of a production

In the simplest form of a production, conditions and actions refer directly to the presence (or absence) of objects in working memory. For example, a production might say:

CONDITIONS: block A is clear; block B is clear
ACTIONS: suggest an operator to move block A ontop of block B

This is not the literal syntax of productions, but a simplication. The conditions of a production may also specify the absence of patterns in working memory. For example, the conditions could also specify that "block A is not red" or "there are no red blocks on the table". But since these are not needed for our example production, there are no examples of negated conditions for now.

D.3.2. Architectural roles of productions

Soar productions can fulll four dierent roles: the three knowledge-retrieval problem-solving functions, and the state elaboration function:

1. Operator proposal

2. Operator comparison

3. (Operator selection is not an act of knowledge retrieval)

4. Operator application

5. State elaboration

A single production should not fulll more than one of these roles (except for proposing an operator and creating an absolute preference for it). Although productions are not declared to be of one type or the other, Soar examines the structure of each production and classi- es the rules automatically based on whether they propose and compare operators, apply operators, or elaborate the state.

D.3.3. Production Actions and Persistence

Generally, actions of a production either create preferences for operator selection, or create/ remove working memory elements. For operator proposal and comparison, a production creates preferences for operator selection. These preferences should persist only as long as the production instantiation that created them continues to match. When the production instantiation no longer matches, the situation has changed, making the preference no longer relevant. Soar automatically removes the preferences in such cases. These preferences are said to have I-support (for "instantiation support"). Similarly, state elaborations are simple inferences that are valid only so long as the production matches. Working memory elements created as state elaborations also have I-support and remain in working memory only as long as the production instantiation that created them continues to match working memory. For example, the set of relevant operators changes as the state changes, thus the proposal of operators is done with I-supported preferences. This way, the operator proposals will be retracted when they no longer apply to the current situation.

However, the actions of productions that apply an operator, either by adding or removing elements from working memory, need to persist even after the operator is no longer selected and operator application production instantiation no longer matches. For example, in placing a block on another block, a condition is that the second block be clear. However, the action of placing the rst block removes the fact that the second block is clear, so the condition will no longer be satisfied.

Thus, operator application productions do not retract their actions, even if they no longer match working memory. This is called O-support (for "operator support"). Working memory elements that participate in the application of operators are maintained throughout the existence of the state in which the operator is applied, unless explicitly removed (or if they become unlinked). Working memory elements are removed by a reject action of a operatorapplication rule.

Whether a working memory element receives O-support or I-support is determined by the structure of the production instantiation that creates the working memory element. Osupport is given only to working memory elements created by operator-application productions.

An operator-application production tests the current operator of a state and modies the state. Thus, a working memory element receives O-support if it is for an augmentation of the current state or substructure of the state, and the conditions of the instantiation that created it test augmentations of the current operator.

When productions are matched, all productions that have their conditions met fire creating or removing working memory elements. Also, working memory elements and preferences that lose I-support are removed from working memory. Thus, several new working memory elements and preferences may be created, and several existing working memory elements and preferences may be removed at the same time. (Of course, all this doesn't happen literally at the same time, but the order of rings and retractions is unimportant, and happens in parallel from a functional perspective.)

D.4. Preference memory: Selection Knowledge

The selection of the current operator is determined by the preferences in preference memory. Preferences are suggestions or imperatives about the current operator, or information about how suggested operators compare to other operators. Preferences refer to operators by using the identier of a working memory element that stands for the operator. After preferences have been created for a state, the decisión procedures evaluates them to select the current operator for that state.

For an operator to be selected, there will be at least one preference for it, specically, a preference to say that the value is a candidate for the operator attribute of a state (this is done with either an

"acceptable" or "require" preference). There may also be others, for example to say that the value is "best".

D.4.1. Preference semantics

Only a single value can be selected as the current operator, that is, all values are mutually exclusive. In addition, there is no implicit transitivity in the semantics of preferences. If A is indierent to B, and B is indierent to C, A and C will not be indierent to one another unless there is a preference that A is indierent to C (or C and A are both indierent to all competing values).

- **Acceptable (+)** An acceptable preference states that a value is a candidate for selection. All values, except those with require preferences, must have an acceptable preference in order to be selected. If there is only one value with an acceptable preference (and none with a require preference), that value will be selected as long as it does not also have a reject or a prohibit preference.
- **Reject (-)** A reject preference states that the value is not a candidate for selection.
- **Better (>), Worse (<)** A better or worse preference states, for the two values involved, that one value should not be selected if the other value is a candidate. Better and worse allow for the creation of a partial ordering between candidate values. Better and worse are simple inverses of each other, so that A better than B is equivalent to B worse than A.
- **Best (>)** A best preference states that the value may be better than any competing value (unless there are other competing values that are also "best"). If a value is best (and not rejected, prohibited, or worse than another), it will be selected over any other value that is not also best (or required). If two such values are best, then any remaining preferences for those candidates (worst, indifferent) will be examined to determine the selection. Note that if a value (that is not rejected or prohibited) is better than a best value, the better value will be selected. (This result is counterintuitive, but allows explicit knowledge about the relative worth of two values to dominate knowledge of only a single value. A require preference should be used when a value must be selected for the goal to be achieved.)
- **Worst (<)** A worst preference states that the value should be selected only if there are no alternatives. It allows for a simple type of default specication. The semantics of the worst preference are similar to those for the best preference.
- **Indifferent (=)** An indifferent preference states that there is positive knowledge that it does not matter which value is selected. This may be a binary preference, to say that two values are mutually indierent, or a unary preference, to say that a single value is as good or as bad a choice as other expected alternatives. When indifferent preferences are used to signal that it does not matter which operator is selected, by default, Soar chooses randomly from among the alternatives.
- **Numeric-Indifferent (= number)** A numeric-indifferent preference is used to bias the random selection from mutually indierent values. This preference includes a unary indierent preference, so an operator with a numeric-indifferent preference will not force a tie impasse. When a set of operators are determined to be indierent based on all other asserted preference types and at least one operator has a numericindi erent preference, the decisión mechanism will choose an operator based on their numeric-indierent values and the exploration policy. When a single operator is given multiple numeric-indierent preferences, they are either averaged or summed into a single value based on the setting of the numeric-indifferent-mode command. Numeric-indierent preferences that are created by RL rules can be adjusted by the reinforcement learning mechanism. In this way, it's possible for an agent to begin a task

with only arbitrarily initialized numeric indierent preferences and with experience learn to make the optimal decisions.

- **Require (!)** A require preference states that the value must be selected if the goal is to be achieved.
- **Prohibit (~)** A prohibit preference states that the value cannot be selected if the goal is to be achieved. If a value has a prohibit preference, it will not be selected for a value of an augmentation, independent of the other preferences.

If there is an acceptable preference for a value of an operator, and there are no other competing values, that operator will be selected. If there are multiple acceptable preferences for the same state but with dierent values, the preferences must be evaluated to determine which candidate is selected.

If the preferences can be evaluated without con ict, the appropriate operator augmentation of the state will be added to working memory. This can happen when they all suggest the same operator or when one operator is preferable to the others that have been suggested.

D.5. Soar's Execution Cycle: Without Substates

The execution of a Soar program proceeds through a number of cycles. Each cycle has ve phases:

1. **Input:** New sensory data comes into working memory.
2. **Proposal:** Productions fire (and retract) to interpret new data (state elaboration), propose operators for the current situation (operator proposal), and compare proposed operators (operator comparison). All of the actions of these productions are I-supported. All matched productions re in parallel (and all retractions occur in parallel), and matching and ring continues until there are no more additional complete matches or retractions of productions (quiescence).
3. **Decision:** A new operator is selected, or an impasse is detected and a new state is created.
4. **Application:** Productions re to apply the operator (operator application). The actions of these productions will be O-supported. Because of changes from operator application productions, other productions with I-supported actions may also match or retract. Just as during proposal, productions re and retract in parallel until quiescence.
5. **Output:** Output commands are sent to the external environment.

The cycles continue until the halt action is issued from the Soar program (as the action of a production) or until Soar is interrupted by the user.

D.6. Impasses and Substates

When the decisión procedure is applied to evaluate preferences and determine the operator augmentation of the state, it is possible that the preferences are either incomplete or inconsistent. The preferences can be incomplete in that no acceptable operators are suggested, or that there are insucient preferences to distinguish among acceptable operators. The preferences can be inconsistent if, for instance, operator A is preferred to operator B, and operator B is preferred to operator A. Since preferences are generated independently, from dierent production instantiations, there is no guarantee that they will be consistent.

D.6.1. Impasse Types

There are four types of impasses that can arise from the preference scheme.

- **Tie impasse:** A tie impasse arises if the preferences do not distinguish between two or more operators with acceptable preferences. If two operators both have best or worst preferences, they will tie unless additional preferences distinguish between them.
- **Conflict impasse:** A con ict impasse arises if at least two values have con icting better or worse preferences (such as A is better than B and B is better than A) for an operator, and neither one is rejected, prohibited, or required.
- **Constraint-failure impasse:** A constraint-failure impasse arises if there is more than one required value for an operator, or if a value has both a require and a prohibit preference. These preferences represent constraints on the legal selections that can be made for a decisión and if they con ict, no progress can be made from the current situation and the impasse cannot be resolved by additional preferences.
- **No-change impasse:** A no-change impasse arises if a new operator is not selected during the decisión procedure. There are two types of no-change impasses: state no-change and operator no-change:
  - **State no-change impasse:** A state no-change impasse occurs when there are no acceptable (or require) preferences to suggest operators for the current state (or all the acceptable values have also been rejected). The decisión procedure cannot select a new operator.
  - **Operator no-change impasse:** An operator no-change impasse occurs when either a new operator is selected for the current state but no additional productions match during the application phase, or a new operator is not selected during the next decisión phase.

There can be only one type of impasse at a given level of subgoaling at a time. Given the semantics of the preferences, it is possible to have a tie or con ict impasse and a constraintfailure impasse at the same time. In these cases, Soar detects only the constraint-failure impasse.

The impasse is detected during the selection of the operator, but happens because one of the other four problem-solving functions was incomplete.

D.7. Learning

When an operator impasse is resolved, it means that Soar has, through problem solving, gained access to knowledge that was not readily available before. Therefore, when an impasse is resolved, Soar has an opportunity to learn, by summarizing and generalizing the processing in the substate.

One of Soar's learning mechanisms is called chunking; it attempts to create a new production, called a chunk. The conditions of the chunk are the elements of the state that (through some chain of production rings) allowed the impasse to be resolved; the action of the production is the working memory element or preference that resolved the impasse (the result of the impasse). The conditions and action are variablized so that this new production may match in a similar situation in the future and prevent an impasse from arising.

Chunks are very similar to justications in that they are both formed via the backtracing process and both create a result in their actions. However, there are some important distinctions:

1. Chunks are productions and are added to production memory. Justications do not appear in production memory.

2. Justications disappear as soon as the working memory element or preference they provide support for is removed.

3. Chunks contain variables so that they may match working memory in other situations; justications are similar to an instantiated chunk.

D.8. Input and Output

Many Soar users will want their programs to interact with a real or simulated environment. For example, Soar programs may control a robot, receiving sensory inputs and sending command outputs. Soar programs may also interact with simulated environments, such as a flight simulator. Input is viewed as Soar's perception and output is viewed as Soar's motor abilities.

When Soar interacts with an external environment, it must make use of mechanisms that allow it to receive input from that environment and to eect changes in that environment; the mechanisms provided in Soar are called input functions and output functions.

Input functions add and delete elements from working memory in response to changes in the external environment.

Output functions attempt to eect changes in the external environment.

Input is processed at the beginning of each execution cycle and output occurs at the end of each execution cycle.

Anexo E

Clases y métodos de CERA-CRANIUM y CCBotSoar

En este anexo se resumen las clases y métodos más importantes utilizados para la nueva implementación de la arquitectura CERA-CRANIUM y del CCBotSoar y pretende servir como manual de referencia para una mejor comprensión de la estructura del agente.

E.1. CCBotSoar

Clase principal del bot.

E.1.1. Eventos

Enventos que maneja la clase CCBotSoar. Todos ellos envían información a SinglePercept para su posterior procesamiento.

- **BumpedHandler:** Detección del choque.
- **itemPickedUp:** Detección de Recogida de objeto.
- **hearNoise:** Detección de Sonido de alrededor.
- **playerAppeared:** Detecta cuando un jugador entra dentro del campo visual.
- **playerUpdated:** Este evento recarga la posición del jugador que es visible. (Después de haber sido detectado por el evento "playerAppeared")
- **PlayerFirstEncountered:** Guarda los datos de los jugadores que encuentra pero solo una vez por jugador.
- **PlayerDestroyed:** Destruye el jugador cuando un jugador se desconecta. (Error en la ejecución de este evento (No se ejecuta))

Funciones

Funciones utilizadas por la clase CCBotSoar

- **prepareBot:** Se inicializa el bot y se prepara toda la estructura para el razonamiento.
- **getInitializeCommand:** Inicializamos el bot con un traje y nombre(nick).
- **botInitialized:** Inicializamos los rayos que irá creando el bot para la búsqueda de intersecciones.
- **botSpawned:** Indica que hemos aparecido en el mundo en la consola del bot.
- **resetLastKnowInformation:** Resetea toda la información del bot tanto posición como armas...
- **logic:** Desde aquí detectamos parte de la percepción : Donde se mira, donde se va a mirar, la vida de la cual disponemos y el momento en que cambia, los rayos para determinar posibles caminos, el arma que se tiene seleccionada, la munición y los objetos visibles. Y al final ejecutamos una acción con todos esos datos. Los datos anteriores se envían a SinglePercept.
- **botKilled:** Detectamos la causa de la muerte y se lo enviamos a SinglePercept y reseteamos los valores anteriores con resetLastKnowInformation.
- **main:** Detecta si se quiere el bot para un servidor online o local y Genera el bot y lo inicializa.
- **submitPhysicalPercept:** Recibe la información a procesar y la envía a la correspondiente función para que ella se encargue de enviar la información a las correspondientes funciones.
- **getTime:** Función que devuelve el tiempo del bot.

E.2. Clases extra (*conscious.robots.extra*)

FileManager: Clase encargada de leer/escribir todos los archivos externos que se quieran utilizar (principalmente para escribir los archivos Soar para la memoria a largo plazo).

E.2.1. Risk

Clase que gestiona la variable Riesgo para la modificación del radio de selección de activación en el *workspace*.

[Max/Min/Next/Prev]Coef: Pesos para el cálculo del riesgo.

CalculateRisk (Game game, AgentInfo info): Calcula el valor de la variable Riesgo obteniendo todas las puntuaciones necesarias.

E.3. Acciones (*conscious.robots.actions*)

E.3.1. Action

Priority: Crea un sistema de prioridades para detectar cual es más importante.

Iindex: Indice de la acción(coordenadas obsolutas).

CreationTime: Indicamos el tiempo de creación de la acción.

ExecTime: Cogemos el tiempo de ejecución.

Activation: Indicamos el nivel de activación.

Compare: Compara dos acciones dependiendo de su prioridad.

ActionFactoryInterface: Crea una interfaz para agrupar las distintas funcionalidades de un objeto en subconjuntos más manejables. (Todos ellos devuelven SimpleAction).

E.3.2. SimpleAction (extensión de *Action*)

Type: STRAFE_TO, MOVE, LOOK_AT, MOVING_LOOKING, SHOOT, STOP_SHOOTING, SELECT_WEAPON, JUMP, NOOP

CreateSelectWeapon **CreateLookAtAction** **CreateMoveAction**
CreateJumpAction
CreateStrafeToAction **CreateStopShootingAction** **ActionPriorityComparator**
CreateMovingLookingAction **CreateShootAction**

- **JumpAction:**

  Ejecutar un salto o un doble salto.

- **MovingLooking:** Mover y mirar a la vez.
- **SelectWeapon:** Enviamos el arma que tenemos seleccionada.
- **ShootAction:** Disparar. Podemos elegir disparo primario o secundario.
- **StopShootingAction:** Parar de disparar.

E.3.3. Acciones complejas (extensión de *Action*)

Type: GO_TO, MOVE_TO, LOOK_AT, SHOOT_TO, STOP_SHOOT, SHOOT_ENEMY, MOVE_CLOSER.

- **LookComplexAction:** Mirar a la acción más compleja
- **MoveCloserComplexAction:** Moverse cerca del lugar de la acción compleja.
- **MoveComplexAction:** Moverse a hacia la acción más compleja.
- **MovementPlannerComplexAction:** Movimiento planeado según la acción compleja
- **ShootEnemyComplexAction:** Indica la posición, nombre y prioridad del enemigo.
- **StrafeToAction:** Destrozar a tiros. getFocus: Tiene la posición del foco al que nos estamos moviendo.

E.4. Razonamiento CERA (*conscious.robots.cera*)

Jindex: Calculos de coordenadas, distancias, rotaciones...

Proprioception: Información interna del cuerpo (AGENT_COLLISION_RADIUS, radio de colisión del agente con el resto de elementos del juego).

ActionPriorityComparator: Compara dos acciones dependiendo de su prioridad.

CeraStats: Crea los nombres de las capas.

Cera: Clase que define las características de CERA.

- **LogInfo/LogError:** Información al archivo de log.

E.4.1. CeraContext

- **J:** Localización relativa.
- **T:** Tiempo general.
- **Relevance:** Relevancia del contexto.
- **ActivationTime:** Tiempo en el que fue activado.

E.4.2. CeraLayer

- **setParentLayer:** Prepara las capas superiores.
- **RegisterProcessor:** Coloca los procesos que van a la zona de trabajo de la capa.
- **GetWorkspace:** Recoge las capas de la zona de trabajo.
- **SubmitPercept:** Envia la percepción a la zona de trabajo de la capa. SubmitPercepts: Envía una lista de percepciones a la capa.
- **SubmitAction:** Sera sobre escrita por otra función. SubmitActions: Sera sobre escrita por otra función.
- **IsItTimeTo:** Indica que es tiempo de hacer una acción base según el tiempo.
- **getChildLayer:** Recoge las referencias de los hijos de la capa.

E.4.3. CeraCore (extensión de CeraLayer)

- **IncrementTick:** Aumenta el reloj del nuecleo.
- **Process[Control/Control]Percept:** Procesa los percepciones de control y misión.
- **ProcessA[Novelty/Restart]:** Procesa una novedad o un nuevo contexto de reinicio.
- **ResetLayers:** Reinicia la capa principal.
- **SwitchContext:** Cambia el contexto actual por una localización indicada de percepción.
- **SendContextCmd:** Envía un comando que activa las capas inferiores.

- **DiscardOldPercepts:** Elimina las percepciones antiguas.
- **DiscardLessActivePercepts:** Elimina las percepciones con menos actividad.
- **SendCommand:** Envia un comando informando de un excepción porque no se puede soportar.

E.4.4. CeraException

- **CeraException:** Constructor de una excepción.
- **CeraException:** Mensaje de excepción.

E.4.5. CeraMission (extensión de CeraLayer)

- **CeraMission:** Constructor que crea capas de misisones.
- **IncrementTick:** Incrementa el tiempo de reloj de esta capa.
- **getTick:** Recoge el tiempo.
- **SubmitSimpleAction:** Devuelve el canal más bajo asociado a la zona de trabajo.
- **SubmitComplexAction:** Devuelve el canal más bajo asociado a la zona de trabajo.
- **SendCommand:** Envía el comando indicado a la zona de trabajo.
- **ResetLayer:** No hay nada que resetear.

CeraPhysical: sistemas sensitivos y motores específicos.

- **setMissionLayer:** Prepara la capa superior (misión).
- **IncrementTick:** Prepara el reloj de la capa física.
- **DiscardOldActions:** Desecha las acciones complejas.
- **PurgeExecQueue:** Elimina acciones en ejecución de la cola que tienen poca prioridad.
- **GenerateTimePercept:** Genera el tiempo de percepción y lo envía a la zona de trabajo.
- **getTick:** Recoge el tiempo de la capa física.
- **SubmitComplexAction:** Envía una acción compleja a la zona de trabajo de la capa.
- **getNextActionsSoar:** Carga en el sistema Soar las acciones propuestas y recoge la elegidad por el sistema de reglas (utiliza la función **avanceSoar** y también la función **actualizarEstadoBot**).
- **getNextActionsOnePerType:** Devuelve una colección de acciones a ejecutar (funcionamiento previo del CCBot2).
- **SendCommand:** Envía un comando especificado a la zona de trabajo y a la capa.
- **ResetLayer:** Reiniciamos la capa física.

E.4.6. Functions

- **Gaussian:** Calculas las funciones gaussian.
- **TimeDiffWeighting:** Calcula una anchura según las funciones de gaussian.
- **SpatialDiffWeighting:** Calcula una anchura según las funciones de gaussian.

E.4.7. IA

- **actualizar[Input/Output]:** actualiza el vector de entrada de Soar.
- **insertAction:** introduce una acción en el sistema Soar.
- **reglaRefuerzo:** actualiza el archivo externo en el que se almacenan todas las nuevas reglas de selección creadas.
- **[put/valor]Input:** actualiza/devuelve el valor de una variable del vector de entrada.
- **stepSoar:** avanza un número concreto de fases del ciclo de Soar.
- **escucharEventosSoar:** recibe y procesa los eventos de entrada/salida de Soar.

E.4.8. DeathMap

Clase que contiene la información sobre el Mapa de Peligro.

- **ProcessPercept:** procesa los perceptos de muerte para actualizar el mapa de peligro.
- **updateDeathMap:** actualiza el mapa de peligro incrementando el nivel (*PLUS_OWN_DEATH*) de peligro en el punto indicado.
- **expand:** aumenta el nivel (*PLUS_OTHER_DEATH*) de los puntos cercanos cuya distancia es menor que un radio definido (*radiusExpand*)
- **dangerousPoint:** devuelve el nivel de peligro en un

E.4.9. State

Contiene información sobre el Estado del bot y provee de las funciones necesarias para su gestión.

Campos: **Salud, Munición, VeoSalud, VeoMunición, VeoEnemigo, MeAtacan, EstoyMuerto, Motivación** y **Peligro**

- **ProcessPercept:** procesa todos los perceptos interesantes para actualizar el estado del bot.

E.4.10. Motivation

- **actualizarBienestar:** actualiza el bienestar como el bienestar máximo menos el valor de la suma de todos los drives.
- **actualizarDrives:** actualiza los drives en función del tiempo y del nivel de la variable asociada.
- **motivacionDominante:** devuelve la motivación dominante.
- **satisfacer[Municion/Salud/Violencia]:** reduce al mínimo el valor del drive como efecto de un evento concreto.

E.5. CRANIUM (*conscious.robots.cranium*)

Activation: Nivel de activación de CRANIUM.

Confidence: Nivel de confianza.

PerceptActivationComparator Compara dos percepciones dependiendo de su nivel de activación

ProcessResult Coge las acciones y perceptos generadas en un procesador.

PerceptNotification: Notificaciones de la percepción.

StaticConfig: Variables estáticas de configuración.

WScommand: Comandos que pueden ser enviados a la zona de trabajo para cambiar su funcionamiento.

- **ResetCmd**: Resetea los parámetros del bot, normalmente cuando lo han matado.
- **ContextCmd:** Sirve para preservar los valores de los objetos.

E.5.1. Workspace

- **addPerceptListener:** Notificaciones de la percepción
- **IncrementTick:** Contador que nos indica cuando debemos deshacernos de percepciones o acciones.
- **DiscardOld[Percept/Action]Data:** Desecha las percepciones/acciones antiguas.
- **PurgeSTMPerceptQueue:** Impide que se exceda la Memoria a corto plazo (STM) quitando el exceso de información. (Revisar)
- **PurgeSTMActionQueue:** Impide que se exceda la Memoria a corto plazo (STM) quitando el exceso de información. (Revisar)
- **SendCommand:** Se encarga de la mandar los comandos que se van a ejecutar en la zona de trabajo.
- **SubmitPercept:** Envía la percepción a la zona de trabajo.
- **Submit[Complex/Simple]Action:** Envía una acción compleja a la zona de trabajo.
- **CalculateActivation:** Calcula el nivel de activación de la acción.
- **getCurrentFocusJ:** Devuelve la localización del foco activo
- **TakeCurrentWinner[Percept/Action]:** Coge y borra la percepción/percepto que está cerca del contexto activo.
- **WithinFringe:** Indica si una acción esta dentro del foco o no.
- **TakeFringe[Actions/Percepts]:** Coge y borra una lista de acciones/percepciones que están dentro de la franja.
- **IsItTimeTo:** Indica si es el momento para ejecutar una acción en función del tiempo suministrado.

E.6. Percepción (*conscious.robots.percepts*)

IPerceptListener: Crea una interfaz para agrupar las distintas funcionalidades de un objeto en subconjuntos más manejables.

- <[Single/Complex/Mission/Control]Percept.Type> **Get[Input/Output][Single/Complex/Mission/Contr**
- <ComplexAction.Type> **GetActionInputTypes()**
- void **Add[Input/Output][Single/Complex/Mission/Control]PerceptType([Single/Complex/Mission t)**
- void **AddOutput[Simple/Complex]ActionType(SimpleAction.Type t)**
- **ProcessResult ProcessPercept(AbstractPercept percept)**
- **ProcessResult ProcessComplexAction(ComplexAction action)**

PerceptActivationComparator: Compara dos percepciones según su prioridad.

PerceptEvent: Evento de percepción.

E.6.1. AbstractPercept

- **Activation:** Nivel de activación de la percepción.
- **Confidence:** Coge el nivel de confianza de la percepción.
- **Nature:** Naturaleza de la percepción.
- **ComingFrom:** Lugar de donde ha venido la percepción.
- **Time:** Tiempo en el que la percepción ha sido creada.
- **Jindex:** Posición relativa de la percepción.
- **Source:** Datos del código de la percepción.
- **JAbs:** Posición absoluta de la percepción.
- **JAgent:** Posición del agente cuando sepamos la posición de esta percepción.
- **setAsSensorReading:** Lectura de sensores, identificando su tipo (exteroceptiva, o del mundo exterior, o propioceptiva, del propio agente).
- **setAsProcessorOutput:** Cambia las percepciones a procesos con la confianza correspondiente e identificando el tipo de sensor.

E.6.2. SinglePercept (extensión de AbstractPercept)

Type: TIME, BUMP, LOOKING_AT, LOOK_AT, SHOOT_AT, NO_SHOOT, SHOOTING, HEALTH, KILLED, PLAYER_FIRSTENCOUNTERED, PLAYER_APPEARED, PLAYER_DISSAPPEARED, PLAYER_UPDATED, PLAYER_DESTROYED, BEING_DAMAGED, RAYCASTING, OBSTACLE, GOT_STUCK, MOVING_TO, NOT_MOVING, AGENT_LOCATION, ITEM, ITEM_PICKED, CURRENT_WEAPON, AMMUNITION, LOCATION_REACHED, LOCATION_CLOSE, HEARD_NOISE

- **AmmunitionPercept:** Constructor que indica la munición de las armas.
- AbsolutePointPercept:
- **BeingDamagedPercept:** Constructor que indica la percepción del daño que te hacen al atacarte.
- **BumpedPercept:** Constructor que se genera cuando el bot se choca contra un enemigo.
- **CloseToTargetLocation:** Constructor que indica que estamos cerca a la posición donde va a estar el enemigo.
- **GotStuckPercept:** Constructor para detectar si hemos sido atacados.
- **HealthPercept:** Constructor que indica la vida del bot.
- **HearNoisePercept:** Constructor que indica si se ha escuchado algo.
- **ItemPercept:** Constructor que indica que hay un objeto.
- **ItemPickedUpPercept:** Constructor que indica si hemos recogido el objeto.
- **KilledPercept:** Constructor que indica si hemos muerto y las causas (WORLD, BULLET, VEHICLE, UNKNOWN)
- **LocationReachedPercept:** Constructor que indica hasta donde se ha alcanzado la percepción.
- **LookAtPercept:** Constructor que indica donde deberíamos mirar.
- **LookingPercept:** Constructor que indica donde estamos mirando.
- **MovingPercept:** Constructor que indica hacia donde nos movemos.
- **NotMovingPercept:** Constructor que indica la parada del bot.
- **ObstaclePercept:** Constructor que indica cuando hay un obstáculo.
- **RaycastingPercept:** Constructor que indica la dirección a moverse.
- **SelectedWeaponPercept:** Constructor que indica el arma actual.
- **ShootPercept:** constructor que indica donde deberíamos disparar.(No se usa, utiliza una acción compleja en su lugar)
- **ShootingPercept:** constructor que indica si estamos disparando.
- **TimePercept:** Constructor que indica el paso del tipo. TimePercept: Constructor con sobrecarga que indica el tiempo transcurrido.
- **PlayerPercept:** Constructor que indica todos los datos de un jugador.
  - **getName:** Coge el nombre del jugador.
  - **isVisible:** Coge el indicador de si lo tenemos a la vista
  - **isFiring:** Indica si está disparándonos.
  - **getRotation:** Indica la rotación en la que esta.
    - **PlayerAppearedPercept:** Constructor que indica cuando un jugador aparece delante.

- **PlayerDestroyedPercept:** Constructor que indica que un jugador se ha ido del mundo.
- **PlayerDissappearedPercept:** Constructor que indica cuando hemos perdido de vista a un jugador. Constructor sobrecargado que indica a un jugador que ya habíamos visto y ha desaparecido.
- **PlayerFirstEncounteredPercept:** Constructor que indica los datos de un jugador la primera vez que lo vemos.
- **PlayerUpdatedPercept:** Constructor que indica cuando un jugador ha cambiado de estado, de arma, posición...

E.6.3. ComplexPercept (extensión de AbstractPercept)

Type: PLAN_FINISHED, PLAN_CLOSE, UNKNOWN

- **ClosePlanFinishComplexPercept:** Constructor que cierra el plan complejo cuando esta cerca de finaliza.
- **PlanFinishedComplexPercept:** Constructor que indica cuando a terminado el plan.

E.6.4. ControlPercept (extensión de AbstractPercept)

Cause: Obtiene la percepción de origen que desencadenó la condición de control.

Type: MATCH, MISMATCH, NOVELTY, RESTARTED

- **MatchPercept:** Constructor que indica que todo va según lo esperado.
- **MismatchPercept:** Constructor que indica que algo no va según lo esperado.
- **NoveltyPercept:** Constructor que indica cuando hay una algo nuevo que no está programado.
- **RestartedPercept:** Constructor que reinicia todas las percepciones.

E.6.5. MissionPercept (extensión de AbstractPercept)

Type: ENEMY_ATTACKING, UNKOWN_ATTACKING

- **EnemyMPercept:** Constructor que indica que ha aparecido un enemigo.
- **SomeoneUnkowAttackingPercept:** Constructor que indica que alguien le está atacando, sin saber de dónde.

E.7. Ejecución en Pogamut (*conscious.robots.pogamut*)

Calculate

- **ToRelative:** Calcula la localización relativa de la percepción.
- **ToJindex:** Convierte un punto 3d o una localización de Pogamut en coordenadas de CERA.

- **RotationToJindex:** Transforma coordenadas de rotación en coordenadas de objeto.
- **VelocityToJindex:** Transforma un objeto a de velocidad a coordenadas.
- **ToLocation:** Transforma una localización en una localización de objeto de Pogamut.

Transformadores

- **JumpActionP** (extensión de JumpAction) Constructor que indica la realización de salto.
- **LookAtActionP** (extensión de LookAtAction) Constructor que indica la acción de mirar (Usado pero sin usarse, Revisar)
- **MoveActionP** (extensión de MoveAction) Constructor que indica la acción a donde nos movemos. (Usado pero sin usarse, Revisar)
- **MovingLookingActionP** (extensión de MovingLookingAction) Constructor que indica hacia donde miramos mientras nos movemos.
- **SelectWeaponP** (extensión de SelectWeapon) Constructor que indica el arma elegida.
- **ShootActionP** (extensión de ShootAction) Constructor para disparar con el modo primario / secundario.

PogamutAction: Interfaz para la creación de la acción.

PogamutActionFactory: Métodos para la creación de acciones.

- **Create[SelectWeapon/MovingLookingAction/StopShootingAction/ShootAction/JumpAction]**

StaticProperties: PROPIEDADES DEL JUEGO

- **addNoiseLocation:** Indica la localización del sonido y si el bot se acerca a él.
- **pointAhead:** Ir recto hacia el sonido.
- **getModNoiseValue:** Recoger los valores de la distancia hasta el sonido.
- **isLooking:** Constructor que indica están mirando al bot y cuál es la distancia desde la que lo miran.

E.8. Procesadores (*conscious.robots.processor*)

E.8.1. Processor

Processor: Constructor que prepara las variables e inicia el procesador

setWorkspace: Prepara la zona de trabajo.

setActionFactory: Prepara la interfaz de acciones.

NothingToDo: Indica que no hay datos.

DiscardOldData: Elimina los datos antiguos de la cola.

SubmitResults: Envía las nuevas percepciones o acciones a la zona de trabajo.

run: Elimina los datos antiguos y prepara los datos nuevos.

start: Inicia el proceso de información.

stop: Elimina el proceso de información.

Get[Input/Output][Single/Complex/Mission/Control]PerceptTypes: Listas de perceptos.

GetActionInputTypes: Lista que contiene los tipos de las acciones de entrada.

NotifyPercept: Indica que hay nuevas percepciones.

NotifyComplexAction: Indica que hay nuevas acciones complejas.

Las siguientes funciones, añaden los distintos tipos al procesador si este está dispuesto:

Add[Input/Output][Single/Complex/Mission/Control]PerceptType

AddInputComplexActionType - AddOutput[Simple/Complex]ActionType

ProcessComplexAction: Llama a la siguiente acción compleja para procesarla.

ProcessPercept: Llama a las siguiente percepción para procesarla.

ProcessResult

[get/set][Simple/Complex]Actions: Recoge/prepara prepara la lista de acciones (simples o complejas).

[get/set]Percepts: Recoge/prepara las percepciones.

ProcessResult: Constructor que crea las listas de acciones y percepciones.

add[Simple/Complex]Action: Añade una nueva acción.

addPercept: Añade una nueva percepción.

E.8.2. Lista de procesadores

- **AttackDetector:** Indica que nos atacan cuando la vida nos baja.
- **AttackingMovement:** Indica los movimientos de ataque.
- **AvoidObstacle:** Indica que hay un obstáculo.
- **BackupReflex:** Crea una copia de golpear y destrozar a tiros.
- **ChasePlayer:** Indica que vamos a perseguir a un enemigo.
- **CloseTargetLocation:** Indica que tenemos un enemigo cerca.
- **Curiosity:** Indica que tenemos curiosidad y vamos a observar
- **EnemyDetector:** Indica que se ha detectado un enemigo.
- **GazeGenerator:** Indica si hemos visto algo.
- **GetVisibleItem:** Indica los objetos visibles.
- **GoToMalignusPoint:** Indica que vamos a un punto maligno.

- **JumpObstacle:** Indica un obstáculo a saltar.
- **KeepEnemiesFar:** Indica que nos tenemos que mover a un lugar lejos de enemigos.
- **TooCloseToPlayers:** Estamos cerca de un enemigo, debemos correr.
- **LocationReached:** Indica si hemos alcanzado el punto al que queríamos llegar.
- **MoveCloserPosition:** Indica que nos movemos cerca de un objetivo.
- **MoveLooking:** Indica hacia donde nos miramos al movernos.
- **MoveToPoint:** Indica el moverse a un punto, según un plan establecido.
- **NotMoving:** Indica que no nos movemos al mismo punto de antes.
- **ObstacleDetector:** Detecta si hay obstáculos.
- **ObstacleDetectorNR:** Constructor que detecta si hay obstáculos.
- **PlayerDissappearDetector:** Constructor que indique que ha desaparecido un jugador.
- **PlayersNoveltyDetector:** Constructor que indica que algo no programado a ocurrido.
- **RandomJumpGenerator:** Constructor que indica un salto aleatorio con un 1 % de probabilidad cada 5 minutos.
- **RandomMove:** Constructor que indica un movimiento aleatorio en caso de chocarnos contra algo.
- **RestartDetector:** Constructor que indica el reseteo de todo cuando hemos muerto.
- **RunAwayFromPlayers:** Constructor que inicia todos los parámetros parra huir del jugador.
- **SelectBestWeapon:** Constructor que indica el mejor arma del inventario.
- **SelectEnemyToShoot:** Constructor que indica a quien queremos disparar.
- **ShootEnemy:** Indica que disparamos a un enemigo.
- **ShootTo:** Elige a quien disparar y se disparar.
- **StuckDetector:** Indica que nos hemos chocado.
- **RememberItems:** Almacena la posición de los objetos de la partida e incita a ir hacia ellos cuando los niveles de salud y munición son bajos.
- **Wander:** Genera acciones de movimiento continuamente.

E.9. Entrenamiento (training)

Este paquete contiene varios tipos de bots predefinidos para utilizarlos en el entrenamiento del bot.

Anexo F

Tecnologías utilizadas

En este capítulo se detallan las tecnologías utilizadas durante el desarrollo de este proyecto y las ventajas y desventajas de su uso.

F.1. JSoar

JSoar es una implementación en Java de Soar. Soar es una arquitectura cognitiva para el desarrollo de los sistemas que exhiben un comportamiento inteligente. JSoar es esencialmente una biblioteca para la construcción de sistemas basados en agentes aunque se tiene que escribir tanto el código Soar para definir el comportamiento del agente como el código Java para conectar el agente al entorno, los interfaces, etc. Por tanto, aunque JSoar es una implementación de Soar, para utilizar esta tecnología eficientemente es necesario saber programar en Soar.

Algunas características de JSoar:

- APIs (ver Glosario) en Java.
- Soporta metalenguaje (JRuby, Jython, Rhino (JavaScript), Groovy, Scala, Clojure, etc).
- Código base y herramientas sencillas de utilizar que permiten una rápida iniciación.
- Integración limpia con los sistemas software.

JSoar requiere Java 1.6 para funcionar, pues utiliza características de Java no disponibles en las versiones anteriores a la 1.6.

El API de JSoar está dividido en una serie de clases e interfaces. Aunque esto hace las pruebas y las ampliaciones más sencillas, puede llevar a la confusión y a un código más detallado de lo necesario. Con el mismo espíritu que las clases de "ayuda" de Java, "java.útil.Collections" y "java.útil.Arrays", JSoar ofrece una serie de clases que casi siempre deben tener prioridad sobre las clases homólogas de libre programación por el desarrollador. Son las siguientes:

- **Wmes**: Métodos para buscar y filtrar WMEs (ver Glosario).
- **InputWmes:** Métodos para añadir y actualizar Input WMEs (ver Glosario).
- **Symbols**: Métodos para convertir símbolos en y desde objetos Java.
- **SoarCommands:** Métodos para ejecutar comandos del interprete de Soar.

- **SoarEvents**: Métodos para manejar eventos Soar.

La creación de un proyecto JSoar depende mucho del entorno de trabajo, mecanismo de construcción y de cómo se quiere que encaje con el sistema. Dicho esto, un proyecto estándar de JSoar se puede crear de la siguiente manera:

- Se inicializa el IDE con el que se va a trabajar, en este caso Netbeans.
- Se crea un nuevo proyecto Java.
- Se añade jSoar-core-X.X.X.jar y jSoar-debugger-X.X.X.jar al proyecto.
- Se crea una clase que contenga la función *main* y se crea un agente de JSoar.

En lo referente a las ventajas de esta tecnología, destacamos que ofrece dos tipos de agentes a desarrollar: (1) el "*thread agent*" para programación en diferentes hilos de ejecución, ya que tiene un hilo de ejecución personal y (2) el "*Raw agent*" para programación sin hilos de ejecución; la distribución de JSoar viene con unas librerías Java que incluyen todo el código fuente (esto es útil generalmente para depurar y averiguar lo que hacen los diferentes métodos cuando el javadoc no es suficientemente completo). JSoar dispone también de un *debugger* para comprobar el correcto funcionamiento de los agentes. Sin embargo es necesario señalar que es una tecnología en desarrollo al igual que Pogamut y hay funcionalidades que no funcionan correctamente (esto ocurre, por ejemplo, con el *thread agent* y con el debugger, que sólo funciona con el *thread agent* luego ninguna se ha podido utilizar en este trabajo). Las dudas en el uso de esta tecnología se han resuelto mediante la comunicación vía email con su creador, Dave Ray, con el fin de solventar los problemas que iban surgiendo durante el desarrollo.

F.2. UnrealED

UnrealED (Unreal Editor) es un software incluido en el videojuego que permite la creación y modificación de mapas para el videojuego Unreal Tournament, se ha utilizado este editor para la implementación de los mapas con los que se realizaron las diferentes pruebas y experimentos. Gracias a esta herramienta disponemos de mapas sencillos que se adaptan a las características que requieren nuestros experimentos facilitando así la consecución e interpretación de los resultados. El motivo de la elección de esta herramienta es poder disponer de un método para crear y modificar mapas en el entorno del proyecto.

En lo referente a las ventajas y desventajas, la principal desventaja es que la documentación sobre la herramienta es escasa, aparte de que existe un editor diferente para cada videojuego de la saga Unreal Tournament lo que dificulta todavía más la búsqueda de documentación sobre el manejo de la herramienta. Más en particular, se utiliza un concepto para la creación de objetos que, en vez de generar elementos en el vacío sustrae el elemento que se quiere crear del espacio.